Open Source Verification in an Anonymous Volunteer Network

Peter T. Breuer[a,*], Simon Pickin[b]

[a] *Dept. Comp. Sci., University of Birmingham, Birmingham, UK*
[b] *Dpto. de Sist. Inf. y Comp., Universidad Complutense de Madrid, Madrid, SPAIN*

Abstract

An 'open' certification process is characterised here that is not based on any central agency, but rather on the option for any party to confirm any part of the certification process at will. The model for this paradigm has been a distributed, piece-wise, semantic audit carried out on the Linux kernel source code using a lightweight formal method.

Our goal is a technology that allows open source developers to receive formally backed certifications for their project, in quid pro quo exchanges of resources and expertise with other developers within an amorphous and anonymous cloud of volunteers. To help ensure the integrity of the results, identifying details such as subroutine and variable names are not included in the data sent for analysis, each part of the computation is repeated many times at different sites, and checkpoint information is generated that enables independent checks to be carried out without starting from scratch each time.

Keywords: Formal Methods, Software Verification, Static Analysis, Open Source, Cloud Computing, Distributed Computation

1. Introduction

We have prototyped a distributed platform aimed at the formal verification and static analysis of large open source code bases and the software has been tested using the Linux kernel source code as target. Our approach is motivated by the vision of a future in which a formal verification problem can be sent out to an 'open verification' community of anonymously volunteered resources for resolution. The community ethos means that a problem submitter ought to contribute resources to solving others problems, and volunteers whose machines perform the computations do so partially as quid-pro-quo for their verification problems being solved too. Eric Raymond's arguments in [Ray01] suggest the rationale may be 'selfish altruism' – it is worth a contributor's while to help get

*Corresponding author
Email addresses: ptb@ieee.org (Peter T. Breuer), spickin@fdi.ucm.es (Simon Pickin)

others problems out of the way when that favours the treatment of theirs next.[1] Be that as it may, the community possesses computational resources beyond the reach of an individual, and if its resources are turned on each problem in turn then that makes possible what is not possible for one developer alone.

Hopefully, with the availability of this kind of facility, some supporters of an open source project who do not have the skills to contribute by coding will contribute instead by lending their CPU cycles to certify the releases error-free (with respect to the classes of error being searched for). They might also get involved more deeply in verification themselves, once they see how it works. That it is in people's economic interest to contribute when they come to understand how to make an improvement in an open source context is also explained by Raymond's argument – by contributing they unburden themselves of the maintenance costs of their solution in the face of ongoing changes in the platform; if they do not contribute it or maintain it, with time it loses its worth.

We propose that architecture as a means of removing the bottleneck of a central certification authority or authorities, which occupies a critical position in traditional approaches to certification. An abundance of volunteer CPU cycles allows verification to be repeated several times over for reliability, while stored intermediate results allow each calculation to be rechecked as required. Instead of blind trust, we may rely on many eyes for scrutiny – this is a candidate for an 'open certification' method, in other words. While neither design nor implementation is perfect or complete in our prototype, it is to be hoped that our initiative will stimulate better efforts and further progress towards that goal.

Our prototype system consists of a database back-end and its servers, and Internet clients both running the verification solver software and contributing new problems to the servers. The target community consists of individual developers each with a tiny computing base, which is not continuously online for a multiplicity of reasons. For example, in the experience of the first author, keeping even a four-machine network running at home in 45C temperatures during August (partaking of the experiment reported here) is best avoided.

The Linux kernel code treated in the work reported here is ANSI C [Ame89, Int99] with embedded assembler, and no significant restrictions. There is no inherent limitation to a particular language, however. And while it is the case that (unrestricted) C is a naturally intractable candidate for verification because of indirections via pointers and other infelicitous language features, those obstacles are contemplated in the verification technique used. The technique uses deliberately approximate (but sound) verification logic [BG04, BP06a, BP06b, BP06c, BP06d, BPL06, BP07] customised for each particular verification.

[1] "J. Random Hacker is left with two choices: sit on the patch, or throw it into the pool for free. The first choice ... incurs a future cost – the effort involved in re-merging the patch into the source base in each new release ... The second choice may ... encourage reciprocal giving from others that will address some of J. Random's problems in the future. The second choice, apparently altruistic, is actually optimally selfish in a game-theoretic sense." [Ray01]

1.1. Context and related work

The verification technology used in the work reported here falls in the class of 'lightweight' verification technologies. It is based at the top level on a symbolic programming logic and at the bottom level on decision procedures using mixed integer linear programming [Sch98] implemented using the GNU Linear Programming Kit (GLPK). The GLPK is intended for solving large-scale linear programming and related problems. It is a set of routines itself written in ANSI C and organised as a library, available as part of the GNU Project and is released under the GNU General Public License [Bob01, Gom99]. That is particularly appropriate here because the principal target for our technology has historically been the open source C code of the Linux operating system kernel (see for example [BG04]).

Other lightweight verification technologies in the same class include Splint [EL02] (derived from Larch [GH]), also ESC/Java and Spec# [BKS04]. All these tools make some sacrifices in the area of completeness or precision in order to be useful on the undecorated original source codes, and some require expert annotations to be added to the source. And while the C language is always a particularly difficult target for such technologies, some notable attempts at it have been made.

David Wagner and collaborators in particular have been active in the area (see for example [JW04], where Linux user space and kernel space memory pointers are given different types, so that their use can be distinguished, and [WFBA00], where C strings are abstracted to a minimal and maximal length pair and operations on them abstracted to produce linear constraints on these numbers). That research group often uses model-checking [CES86].

Their approach in [WFBA00, JW04] makes use of both model-checking and abstract interpretation [CC77] (abstraction is used in general in order to 'airbrush away' the more unsavoury aspects of C), and therefore contrasts with contributions like Jeffrey Foster's work with CQual [FTA02], which seek to extend the type system of C in a more controllable direction. In particular, CQual has been used to detect "spinlock-under-spinlock", a sub-case of one of the analyses routinely performed by the tools used in the experiment reported here.

The SLAM project [BR02] originating at Microsoft also analyses C programs using a mixture of model-checking, abstract interpretation and deduction. That technology is intrinsically an order or more of magnitude slower than the basic technology used in the experiment described here, but it also works by creating an abstraction of the program code, and it also generates intermediate state descriptions mechanically.

The Coverity checker [ECCH00] has also been used in the context of the Linux kernel. Coverity is a commercial tool based on a user-extensible version (a *meta-compiler*) of the GNU C compiler, *gcc* [Gri02]. Coverity itself is proprietary, and its innards are not accessible to review, but it may be guessed that the staff of the company can configure into the compiler framework any finite state machine-based computation for the purposes of a custom analysis that they have in mind. It is a less abstract technological solution than the one used here, but shares with it the characteristic of customizability.

Efforts to distribute verification computations to a large number of solvers organised in a well-defined topology are made regularly – see [ABK09], for example. Researchers have particularly sought to distribute model-checking problems onto grid-based machinery. Holzmann defines the notion of 'swarm verification' to describe the technique [HJG08b], adapting the SPIN [Hol03] model-checking tool to the paradigm. In passing, it may be noted that the verification technology used here seems to be part of a recent trend observed by Holzmann in [HJG08a] towards verification of an abstraction of the actual code rather than of a design model. However, the work reported here aims to accommodate the lower performance targets obtainable from zero-cost volunteer CPU cycles available out on the Internet.

Several existing software projects aim to provide infrastructure for the support of so-called 'volunteer computing' -type projects. See for example BOINC [And04, AKW05] from Berkeley. It is not clear at the time of writing if that software would have been a significant aid in our exploratory project, because BOINC clients expect a single data file and return a single result file to the database server, rather than engaging in a continuous interchange, as is the case here. Nevertheless, it may in the future be very helpful in the organisation of the architecture in a full-scale project, particularly in terms of organising permissions for access and classifying the provenance and reliability of the data returned. We sometimes refer to our network of volunteers as a 'cloud', in the sense of an amorphous, anonymous and dynamically varying topology. However, our aim is to provide a volunteer-computing structure similar to that explored in the INRIA project *Clouds@home* (note plural in name) [LKM12, HKA12], in the *Cloud@Home* proposal (note singular in name) [CDPS09], and at CERN [SBA+10], which would bring our use of the term closer to commercial parlance.

Peter Lee [NL96] has approached the problem of automatically checking the trustworthiness of machine code to be executed by an operating system. The idea in *proof carrying* approaches is that incoming code snippets carry a proof that a desired security property is satisfied, and the operating system automatically checks the syntactic relationship of the machine code to the proof. Our approach here is to check the source code instead, although Lee's work presages recent work of the first author of this article setting out a typed assembly language for RISC machine code that guarantees runtime safety properties [BB12].

The present paper follows on from the report in [BP09] by setting the distributed verification trial presented there in the context of a a generic open certification framework. We have abstracted out from that experiment the properties that make it a useful template for an open certification model, and that is what is presented here. The original trial is reported only to the extent that it provides the reader with a concrete foundation on which to site the abstraction.

Regarding the logic-based static analysis technique for imperative programs used here, the logic has been presented previously in [BG04, BP06a, BP06b, BP06c, BP06d, BPL06, BP07, BP09]. In this paper we take the opportunity to (a) present a single logic plus modal operators instead of the three interacting non-modal logics presented in earlier publications, and (b) provide a trace semantics, instead of the predicate-transformer semantics offered in [BP06c],

with the aim of making the technique accessible to readers who prefer models founded in sets and traces to proof systems, and (c) prove the logic sound with respect to our new trace model.

1.2. Contents

This article is organised as follows. Section 2 formally describes the process of certification from the top down in the abstract. After describing certification properties, the section describes how analysis leads to a certification, what certification is and how it may be checked. We show how our analysis calculation is organised to meet the requirements of a part-time volunteer cloud computing context by being conducted piece-wise, with each piece of the calculation being repeated by independent checkers such that an error must be exposed, and we give a category-theoretic characterisation of the situation.

Section 3 succinctly presents the programming logic used in our analysis software in order to provide a self-contained account of the technology, and readers may wish to skip the section if they are not interested in formal logic. A reference account of the trace semantics that gives rise to the program logic is provided in Appendix A, along with a soundness proof, in order to provide solid assurances as to the correctness of the logic.

Section 4 describes the experiment performed on about a million lines of Linux kernel C (and assembler) source code, conducted using monolithic analysis tools in [BP06b] and repeated using a distributed platform in a 'volunteer cloud computing' trial in [BP09]. The configuration of the analysis logic for the experiment is described in Appendix B. The account here serves as an anchoring reference for the abstract view put forward in Section 2.

2. Certification

In this section a formal view of the certification process is set out and related to the procedure implemented in our software. We start by describing in concrete detail what we mean by semantic defects in code, with an example, and explain how false positives for defects may arise when attempting to verify formally that there are none in a piece of code. Our semantic analysis technology is described to a level of detail that we hope enables the reader to make sense of how it all works.

Next, what is meant by a certification process and how what we have is one is described, followed by a description of what is meant by its *accountability*. We describe how our certification process is broken into small pieces to be carried out in different places a bit at a time and how errors or deliberate malversion in the calculation are detected by virtue of that structure. Finally the properties relied on are characterised at the category-theoretic level.

2.1. Notation

'Algebraic ordering' for function composition and application is generally used here. The function f applied to the argument x is written $x\,f$, without

parentheses. When followed by application of function g, the result is written $x\,f\,g$. The composition of f followed by g is written $f\fatsemi g$ and $x\,f\,g = (x\,f)\,g = x\,(f\fatsemi g)$. This convention tends to reduce numbers of parentheses and reading left-to-right gives the application and composition ordering.

If 'analytic ordering' is used for function application, it is indicated by parentheses, thus: $f(x)$ and $g(f(x))$. In this convention, function composition is written $g \circ f$, with $g(f(x)) = (g \circ f)(x)$.

2.2. What is a defect?

A defect d_p is characterised by its kind, d, and locus, p, in the code. For example, 'sleep-under-spinlock$_{120}$' means there is a defect of kind 'sleep-under-spinlock' at line 120. In our system, the presence of a defect d_p is (a) defined by a condition expressed in symbolic logic as $D_p(\mathbf{x})$, and (b) deduced by the formal method to be *possibly* attainable just after point p in the source code. That is, the logical analysis of the source code deduces a post-condition

$$\ldots\ p\ \{\ \mathrm{post}_p\ \}$$

for p and checking the formula using a model-based technique shows there is a non-empty intersection of the post-condition with $D_p(\mathbf{x})$. That is:

$$d_p \quad \Leftrightarrow \quad \exists \mathbf{x}.\ D_p(\mathbf{x}) \wedge \mathrm{post}_p \tag{1}$$

for some values of the logical (i.e. non-program) variables $\mathbf{x}$. The kind of values logical variables are typically used to store are counts like 'the number of unmatched lock or unlock instructions encountered in a trace through the code to this point', as discussed in the example of Subsection 2.3 below.

A detected defect d_p is a formal item in this point of view. It may or may not actually be realized during a particular run of the program, and it may possibly never be realizable at all in any run – see the discussion below in Section 2.4 for an account of unrealisable defects. As to whether it can happen that a defect of type d occurs during a program run at point p but d_p is not flagged by the analysis logic, the answer is "no, provided the analysis logic is sound". This point will be discussed in more detail in Section 2.7. Among other things, it depends on the programming language having been implemented correctly by the compiler and other elements of the tool chain and platform infrastructure, as well as on the language semantics having been abstracted correctly by the analysis logic.

2.3. Example

An example of an interesting defect condition is

$$D_p^{\mathrm{ex}}(x) = \begin{cases} x > 1, & p \text{ a lock call site} \\ x < 0, & p \text{ an unlock call site} \\ \mathrm{false}, & \text{otherwise} \end{cases} \tag{2}$$

where x is a logical variable which counts the number of stacked locks taken in the program. Starting at zero, the variable x is incremented by lock calls and decremented by unlock calls. The pre-/post-condition logic describing x is:

$$\begin{array}{l} \{\phi[x+1/x]\}\ \textbf{lock( )}\ \{\phi\} \\ \{\phi[x-1/x]\}\ \textbf{unlock( )}\ \{\phi\} \end{array} \tag{3}$$

This means that (in the case of **lock**), one expects $x = 1$ to hold after the call when $x+1 = 1$, i.e., $x = 0$, holds before the call. The notation $\phi[x+1/x]$ means 'substitute $x + 1$ for x in formula ϕ'.

At a point p where this defect condition evaluates as feasible (1), it means either that (a) a lock might have been taken twice by that point without a release between the two takes; or (b) a lock may have been released twice by that point without a lock attempt between. These defects d_p^{ex} can by definition (2) only be detected at the sites of a lock or unlock call p. Certification with zero defects in this case means that the code C has been scanned and defects d_p^{ex} have been ruled out. That is, no lock can be taken twice in a row, nor unlocked twice in a row, without an unlock, respectively lock, operation between the two.

2.4. False positives

The checker may flag some defects in some codes via (1) that are, in fact, 'false positives', in that the condition $D_p(\mathbf{x})$ can never be triggered.

Therefore, each of the points in the code p at which a defect of kind d is known to occur, i.e. each of the elements of the list L_d, has to be signed off by the developer as a 'false positive' or 'noted for correction in the next release', or 'noted but no solution yet'. The certificate $X_{L,C}$ certifies that the list is complete, not minimal.

False positives generally fall into two classes. In the first class, a guard condition such as $y^2 < 0$ cannot in practice be breached, but the analysing logic does not know that, and explores a factually impossible code trace as though it were possible. That kind of semantic 'inexactness' is a result of the deliberately approximating nature of the symbolic logic used in any real life analysis. The analysing logic has to be less exact ('more alarmist') than reality or the computation would sometimes never finish in practice.

A typical instance of the second class of false positive arises in the context of the example in Subsection 2.3 above. When two *different* locks are taken in sequence in the code, without an unlock between them, a defect will be detected. The fault here is 'definitional'. The situation is harmless, but it is captured by the defect definition. The problem may be rooted either in the analysis language – different counts for different locks may be difficult to define – or in the analysis logic – which may not be able to distinguish pointer references to different locks in C. The latter is the case here. Different pointers may point to the same lock, or one pointer may point to different locks at different times.

2.5. Technical details of the analysis procedure and our software system

The analysis procedure $\mathcal{A}$ implemented by our software system is organised according to the structure of the source code being analysed. It generates a

pre-/post-condition pair for each program fragment p:

$$\{\text{pre}_p\}\ p\ \{\text{post}_p\}$$

The pair is computed from the pre-/post-condition pairs generated for the component fragments

$$p_i \in p : p = \underset{i}{P}(p_i)$$

where P is the constructor (**if**, **while**, etc.) that produces p from its components p_i. That is

$$(\text{pre}_p, \text{post}_p) = \underset{i}{[P]}(\text{pre}_{p_i}, \text{post}_{p_i}) \quad (4)$$
$$\text{where } \{\text{pre}_{p_i}\}\ p_i\ \{\text{post}_{p_i}\} \text{ for } p_i \in p$$

and $[P]$ is a generator specified in the symbolic logic. It is specified for each syntactic construct of the source language (here C) in the configuration file for the analysis. The reader may refer to Table 1 of Section 3, where the pre- and post-conditions on the bottom of the deduction rule for each constructor can be seen to be composed from the pre- and post-conditions on the top of the rule, to see that the construction shown in (4) is feasible. For the particular configuration defining each generator, the reader may consult Appendix B.

For some constructs, the pre- and post-conditions are fixpoints of a calculation in which an unknown predicate x plays a part, and it must satisfy some condition at node p that is not expressible at the level of the components lower down in the syntax tree. Thus in principle the pre-/post-conditions for components need to be constructed conditionally on x:

$$(\text{pre}_p(x), \text{post}_p(x)) = \underset{i}{[P]}(\text{pre}_{p_i}(x), \text{post}_{p_i}(x))$$
$$\text{where } \{\text{pre}_{p_i}(x)\}\ p_i\ \{\text{post}_{p_i}(x)\} \text{ for } p_i \in p$$

That is handled by instead constructing the pre-/post-conditions with $x = \bot$, then for $x = \bot' > \bot$, then for $x = \bot'' > \bot'$ etc, for a monotonically increasing sequence $\bot, \bot', \bot'', \ldots$ that tends to a fixpoint x (and reaches it after a finite number of iterations - the heuristic/algorithm involved is sure to terminate, but likely to get a fixpoint x that is larger than the least possible). At each iteration, the component structure is copied, so that the copies p_i^j are used for the calculation with $\bot^{(j)}$:

$$(\text{pre}_{p^j}(\bot^{(j)}), \text{post}_{p^j}(\bot^{(j)})) = \underset{i}{[P]}(\text{pre}_{p_i^j}(\bot^{(j)}), \text{post}_{p_i^j}(\bot^{(j)}))$$
$$\text{where } \{\text{pre}_{p_i^j}(\bot^{(j)})\}\ p_i^j\ \{\text{post}_{p_i^j}(\bot^{(j)})\} \text{ for } p_i^j \in p^j$$

The dependencies between the copies are $j_1 < j_2$ implies $p^{j_1} < p^{j_2}$ and $p_i^{j_1} < p_i^{j_2}$ for all i, and as in the copied structure for $j_1 = j_2$, that is $p_{i_1} < p_{i_2} \ \Rightarrow\ p_{i_1}^j < p_{i_2}^j$

and $p_i^j < p^j$. Nodes which depended on the top level structure p in the original now depend on all the copies p^j. In practice, considerable space is saved by copying only the decorations on the syntax tree, rather than copying the syntax tree itself. The sequences leading to a fixpoint are generally quite short – of the order of two or three iterations – and where they are long it is usually an indication of problems in the heuristics that aim at a fixpoint. Nested loops and multiple **goto** labels have the capacity to provoke a combinatorial explosion here. Section 4 reports on how frequently that happens in practice.

We remark that, in principle, if a single defect is being searched for, then no more than n copies need be constructed here, where n is the number of nodes $p' \leq p_i$ in the copied segment. That is because when the analysis for a defect is applied, the defect is eventually detected at between 0 and n of these nodes. It takes up to n analysis computations round the loop involved to find the fixpoint, as the result of the analysis increases monotonically each time. That requires only n copies.

The logic is designed to be sound with respect to the semantics of the target language, in that for each pre-/post-condition pair generated by formula (4), and for each state s of the program that may exist just before p executes and each state s' just after:

$$\mathrm{pre}_p(s) \;\Rightarrow\; \mathrm{post}_p(s') \tag{5}$$

See Appendix A and Theorem AT1 for a proof of soundness.

It is not necessarily the case that pre_p is the weakest precondition that will force post_p, or that post_p is the strongest postcondition forced from pre_p. That means that the symbolic logic generated by the scheme (4) is *approximate* (but sound, following (5)). That gives rise to the name 'symbolic approximation' [BP06c] for the technique. A slightly different approximate symbolic logic is customised for each defect analysis, but each one is sound.

Note that some complexity reduction *is* performed by our tools via lightweight automatic theorem-proving techniques at the stage of producing the tree $T^\dagger$ with the symbolic logic annotations. For example, a formula of the form

$$p \wedge q$$

will be reduced to q if $p \rightarrow q$ is proved on the fly as the formula is generated. Similarly for $p \vee q$. That has proven very effective in reducing complexity. What our tools are not good at is reducing formulae of the general shape $\bigvee_i \bigwedge_j q_{ij}$ to a simpler expression p when there is one, such as in the case of $p \wedge q \vee p \wedge \neg q$. The inadvertent and unrecognised splintering of simple logical expressions into multiple complex cases in this style is the most significant source of the computational explosions that are occasionally encountered during processing. In principle, the situation could be detected and repaired at the checking stage of the process when $T^\ddagger$ is generated (all the atomic propositional forms here are linear inequalities and one could detect when dropping one failed to relax the problem), but that is not done, because the extra computation is usually

prohibitively expensive and apparently only rarely productive in practice (see also Section 4.5).

In the final phase that produces $T^{\ddagger}$, a modelling technique is applied to decide whether

$$\mathrm{post}_p \wedge D_p(\mathbf{x})$$

is satisfiable at any node p of the abstract syntax tree. Since all questions of satisfiability for the predicates in our logic can be reduced to questions of the feasibility of systems of linear inequalities in integer variables, the evaluation $\mathcal{H}$ is performed using mixed linear integer programming. The implementation uses only open source libraries, principally GNU's Linear Programming Kit (GLPK). A non-negative answer by $\mathcal{H}$ to the question asked indicates a *possible* defect d_p of kind d at locus p.

2.6. Certification in the abstract

What certification means in a general context will be described below and further characterised in the next section. Three characteristics stand out:

1. certification is a process and it produces both a *result* and a *certificate*;
2. the certificate has the property that it can be *checked* to have been generated by following the purported process applied to the purported source, generating the purported result;
3. the end result gives certain *guarantees* about the code.

In our system, the process is the computation via formal methods, and the end result is the list of sites in the source code at which certain semantic errors are found. The guarantee is that the source code contains no more errors of the kind searched for. The certificate shows that the claimed process has been applied to the claimed source code and has produced the claimed result.

In the abstract, the certification process is as follows: a process M takes a software code base C and produces an indexed list L of points p in the code at which a defect d is known. Formally (using algebraic order):

$$C\,M = L$$

Let L_d be the set of points in the code at which a defect of kind d is listed in L. While we say that d_p is true if there is a defect of kind d at point p in the code, we also use d_p as an identifier, since the defect of kind d at point p in the code is necessarily unique. That is, 'sleep-under-spinlock$_{120}$' is the name for the one defect of kind 'sleep-under-spinlock' at line 120. Thus:

$$L_d = \{p \mid d_p \in L\}$$

The significance of certification is that a certificate $X_{L,C}$ will 'prove' in a certain public sense that L_d contains all the points p in the code C at which a defect of kind d arises:

$$X_{L,C} \vdash \forall d \in D[L].\ \{p \in C \ \mid \ d_p\} \subseteq L_d$$

where $D[L]$ is the set of defect kinds noted in L (its domain as an indexed list). In practice, L has a header structure naming the elements of the domain – that is, the kinds of defects it treats of – so $D[L]$ is easily extracted from L. Note that it is possible that no defect of type d in C is mentioned in L but d is nevertheless in the domain of L.

It will be the case that the method M is comprehensive enough that if a list L contains the list produced by M applied to code C, then it contains in L_d all the points p in the code at which a defect of kind d arises. That is

$$L \supseteq C\,M \;\;\Rightarrow\;\; \forall d \in D[L].\ \{p \in C \;\mid\; d_p\} \subseteq L_d$$

where $D[L]$ is the set of defect kinds noted in L. We mean by the inclusion that L has the same domain as $C\,M$, and is only larger, if at all, in the number of defects of those kinds d already in the domain of $C\,M$. It may be useful to think of L as a relational database table with one column for the locus p (a primary key) and boolean columns d_1, d_2, ... for each of the defect kinds treated in $C\,M$. Then the inclusion $L \supseteq C\,M$ means that L has the same columns and no fewer rows than $C\,M$.

It suffices then that the certificate $X_{L,C}$ 'prove' to its public that the list L contains the defects found by the method M applied to code C. I.e.

$$X_{L,C} \vdash L \supseteq C\,M$$

So certification means providing a *witness* $X = X_{L,C}$ for the following statement about a list L and the code C:

$$\exists X.\ X \vdash L \supseteq C\,M \tag{6}$$

I.e. code that has more defects than stated in the list does not get a certificate. This means that certificate $X_{L,C}$ has enough information in it to 'prove' that the list has been produced by the method M applied to the code C, which is enough to show that all the defects of the kinds treated in L are accounted for.

2.7. Accountability

It is important for our certification procedure that it can be *checked* that the certificate X produced relates the certified code C and the method M used to certify the defect list L. That is, there is a checking procedure K such that

$$K(X, C, M, L) = \begin{cases} \text{true}, & \text{if } X \vdash L \supseteq C\,M \\ \text{false}, & \text{otherwise} \end{cases} \tag{7}$$

How is that guaranteed in our system?

The answer is: via digital signatures. A digital signature is generated from:

(a) a printout of intermediate results $T^{\ddagger}$ from the analysis M, giving signature $\sigma(T^{\ddagger})$;

(b) the short ASCII file A that configures the analysis, giving signature $\sigma(A)$;

(c) the ASCII file H that expresses the defect condition(s) being scanned for, giving signature $\sigma(H)$;

(d) the list L of noted possible defects, giving signature $\sigma(L)$;

(e) the code C, giving signature $\sigma(C)$;

(f) the file P that configures the code parse, giving signature $\sigma(P)$.

Those signatures comprise X, and enable one to demonstrate that $L \supseteq CM$, as discussed in the following paragraphs.

It is supposed that the developer holds on to a copy of final results $T^{\ddagger}$, a copy of the analysis method configuration $\mathcal{A}$, and a copy of the original code C. Then any part of the certification can be tested at will in case of doubt.

The main ideas involved in the testing procedure K are:

(i) parts of the calculation can be repeated as desired to confirm them;

(ii) in order to be sure that a repeated calculation starts with the right input, the final recorded results can be decomposed as required to provide it;

(iii) to verify that the right results are being checked using the right method applied to the right input, digital signatures are used.

Our method M constructing the list of defects consists first of a parse $\mathcal{P}$ of code C to give a syntax tree T (an initial result):

$$T = C\,\mathcal{P} \tag{8}$$

The analysis $\mathcal{A}$ is applied to the tree T and decorates it with symbolic logic expressions post_p describing the achievable states at each point p in the code according to the symbolic logic, giving the decorated tree $T^{\dagger}$ (an intermediate result):

$$T^{\dagger} = T\,\mathcal{A} \tag{9}$$

Then a checker $\mathcal{H}$ is applied which further decorates the tree with evaluations saying if $D_p \wedge \text{post}_p$ is feasible – that is, that the checker finds some value $\mathbf{x}$ in the intersection, signifying a defect d_p. A defect of kind d is then 'flagged' at p, which means a dth bit is set in a bit-mask in a 'flags' field in the database record representing the abstract syntax tree node p. We may think of this decoration as taking the value 0 or 1 for each defect kind d at each syntax tree node p, and it takes the value 1 for d only at those sites p where a defect d_p is detected. This is a final result:

$$T^{\ddagger} = T^{\dagger}\,\mathcal{H} \tag{10}$$

The list of sites p within the code C at which defects of type d are flagged is constructed from $T^{\ddagger}$. For every node p at which at least one bit, say the dth,

is set in the decoration bit-mask, a defect d_p is entered in the list. This list is supposed to be covered by the list L of known defects:

$$L_d \supseteq \{p \in C : d\text{-bit of flag field on } T^\ddagger \text{ at } p \text{ is set}\} = \{p \in C : d_p\} \tag{11}$$

That is only significant, however, if (A) the checker is accurate and (B) the logic is sound:

(A) accurate checker: for an arbitrary unquantified predicate q in variables $\mathbf{x}$,

$$\mathcal{H}(q) \leq 0 \;\; \Rightarrow \;\; \forall \mathbf{x}.\neg q$$

where $\mathcal{H}(q)$ denotes an estimate by the checker $\mathcal{H}$ of the number of solutions $\mathbf{x}$ satisfying q (here the q of interest is $D_p \wedge \text{post}_p$, where D_p is the defect condition);

(B) sound logic: $\{\mathbf{x} \mid \text{post}_p(\mathbf{x})\}$ is a superset of the values of $\mathbf{x}$ really obtainable in a program run to point p;

Thus if $\mathcal{H}(D_p \wedge \text{post}_p)$ is zero, then, by (A) and (B), D_p is false for every value $\mathbf{x}$ really attainable at point p, and no defect of kind d occurs at point p in practice. That is wonderful news to developer and user alike.

If $\mathcal{H}(D_p \wedge \text{post}_p)$ is positive, then there may be some $\mathbf{x}$ satisfying $D_p \wedge \text{post}_p$, but it may not be one of those $\mathbf{x}$ in post_p that are really achievable in practice. The reported defect d_p represents a 'false positive' in that case. The tighter the post-conditions provided by the analysis logic, the fewer false positives reported.

We have not required that the checker be exact, only that when it says 'no solutions' that it be right, but our checker is exact in that when it gives a positive result $\mathcal{H}(q)$ it is because there is an $\mathbf{x}$ satisfying constraints q, and the more exact a checker is in this sense the fewer the false positives reported.

Suppose that (A), (B) are true, or at least believed true by the community of practitioners. This is a judgement call because it may be that the semantics of the programming language is not completely agreed, or not implemented in agreement with the general interpretation of the standard. In C, for example, the precise timing of the increment $x{+}{+}$ within a compound expression such as $x{+}{+}{+}{+}{+}x$ is notoriously ill-constrained from implementation to implementation and platform to platform, no matter what the standards [Ame89, Int99] say.

Notwithstanding that and other quibbles with respect to practice, provided (11) holds and (A), (B) are generally believed true, the certificate $X_{L,C}$ may be created. It consists of the digital signatures of the code C, the configuration P for the parser $\mathcal{P}$, printed out tree decorations $T^\ddagger$, the configuration A used for the analysis $\mathcal{A}$, the configuration H used for the checker $\mathcal{H}$, and the list L of known defects:

$$X_{L,C} = (\sigma(C), \sigma(P), \sigma(T^\ddagger), \sigma(A), \sigma(H), \sigma(L)) \tag{12}$$

Every step of the procedure that creates the results $T^\ddagger$ can be verified using the signatures in $X_{L,C}$. For example, to get to $T^\ddagger$, one needs to repeat at

least the step (10) that starts from $T^\dagger$. But $T^\dagger$ is just $T^\ddagger$ with some of the decoration dropped. So it can be verifiably obtained from $T^\ddagger$, which is signed. The configuration H of $\mathcal{H}$ is signed and available, and so $\mathcal{H}$ can be applied to verify the derivation of $T^\ddagger$ from $T^\dagger$. Nevertheless, we do not suggest that all the steps be repeated wholesale in order to carry out the verification. The procedure K that checks that X 'proves' that $L \supseteq C\,M$ is rather as follows:

1. *verify* that the data C, P, $T^\ddagger$, A, H offered by the developer and the probative list L in hand matches the given certificate X, i.e., that $X = X_{L,C}$ of (12);
2. *spot-check* the relationships (8-11) that comprise the transformation M of code C to a sub-list of L as detailed further in the following sections.

It is not possible to get away with presenting a different C, P, $T^\ddagger$, A or H as the digital signatures embedded in X will not match.

One may ask if it is possible to short-circuit spot-checks by providing certificates for each of the steps and chaining the certificates. That would not guarantee the semantic relationships – it would be perfectly possible for a malfeasant to produce incorrect initial, intermediate or final results, and certify those.

The more spot-checks carried out, the greater the degree of confidence one may have in the 'proof'. But the fact that checks may be made at all is what makes it risky to attempt to corrupt the certificate process, since discovery may follow. We also advocate repeating elements of the calculation several times over on different computing platforms in order to increase confidence in the honesty of results in the first place.

2.8. Distributing the computation

Our analysis $\mathcal{A}$ and evaluation calculations $\mathcal{H}$ are incremental, stateless, and can be broken off and restarted from the break-off point, as well as repeated partially or wholly. That is the basis for performing the computation in a distributed, piecemeal way and we describe the properties that permit it below. Because the computation has these properties, small 'spot-checks' can be made at any time and will detect errors or deliberate malversion (and correct them) without the whole computation having to be repeated.

Let the constructs p (the nodes and leaves of the abstract syntax tree T produced by the parse) that appear in the code C be $p = \underset{i}{P}(p_i)$ for a syntactic constructor P and components p_i. They define a *dependency* pre-order as the reflexive and transitive closure of the relation:

$$p = \underset{i}{P}(p_i) \quad \Leftrightarrow \quad p_i < p \tag{13}$$

In the pre-order, one code construct 'depends on' (is greater than) another if the second is a component or sub-component, etc., of the first. For example, `if(x<0){ x++; y++; }` depends on the component `x++; y++`, which depends on its component `x++`. The '$\leq$' pre-order is reflexive and transitive.

The operations $\mathcal{A}$ and $\mathcal{H}$ can then be split into steps $\mathcal{A}_p$ and $\mathcal{H}_p$ at $p \in C$:

$$\mathcal{A} = \fatsemi_p \mathcal{A}_p \tag{14}$$

$$\mathcal{H} = \fatsemi_p \mathcal{H}_p \tag{15}$$

The order of the compositions is constrained only by the dependencies $p_i < p$. Formally, operations on different parts of the tree can be performed in any order:

$$\mathcal{A}_{p_1} \fatsemi \mathcal{A}_{p_2} = \mathcal{A}_{p_2} \fatsemi \mathcal{A}_{p_1} \tag{16}$$

$$\mathcal{H}_{p_1} \fatsemi \mathcal{H}_{p_2} = \mathcal{H}_{p_2} \fatsemi \mathcal{H}_{p_1} \tag{17}$$

whenever $p_1 \neq p_2$ and $p_1 \not< p_2$ and $p_2 \not< p_1$ in the dependency relationship. Otherwise, if say $p_1 < p_2$, then the required order is

$$\mathcal{A}_{p_1} \fatsemi \mathcal{A}_{p_2}, \quad \mathcal{H}_{p_1} \fatsemi \mathcal{H}_{p_2}$$

Also, since $\mathcal{A}$ and $\mathcal{H}$ work on different decorative features on the tree, provided the necessary preliminary work has been done in both cases, then analysis and evaluation on different parts of the tree can be done in either order:

$$\mathcal{A}_{p_1} \fatsemi \mathcal{H}_{p_2} = \mathcal{H}_{p_2} \fatsemi \mathcal{A}_{p_1} \tag{18}$$

whenever $p_1 \neq p_2$. When $p_1 = p_2 = p$ then $\mathcal{A}_p \fatsemi \mathcal{H}_p$ is required.

In practice, analysis and evaluation are performed at the same time, because the former is usually computationally cheap relative to the latter. That is, the computation

$$\mathcal{A} \fatsemi \mathcal{H} = \fatsemi_p (\mathcal{A}_p \fatsemi \mathcal{H}_p) \tag{19}$$

is performed. (16, 17, 18) justify the reordering of the components in (19).

That the computation can be broken off and restarted means only that (19) can be further reordered via (16, 17, 18) as

$$\mathcal{A} \fatsemi \mathcal{H} = \fatsemi_{p \in P} (\mathcal{A}_p \fatsemi \mathcal{H}_p) = \fatsemi_{p \in P_1} (\mathcal{A}_p \fatsemi \mathcal{H}_p) \ \fatsemi_{p \in P_2} (\mathcal{A}_p \fatsemi \mathcal{H}_p) \tag{20}$$

where P_1, P_2 is a partition of the set of code fragments $P = P_1 \uplus P_2$ such that

$$p_1 \in P_1 \wedge p_2 \in P_2 \ \Rightarrow \ p_2 \not\leq p_1 \tag{21}$$

I.e., P_1 already contains all the pre-dependencies: $p < p_1 \in P_1 \ \Rightarrow \ p \in P_1$. The set P_1 contains the code fragments that have been completely analysed at the time of the break, and P_2 is the remainder at that time.

Moreover, the computation can be broken off and re-started any number of times. That is, by extending the partitioning to $P = P_1 \uplus \cdots \uplus P_n$ that respects the dependency order:

$$i < j \wedge p_i \in P_i \wedge p_j \in P_j \ \Rightarrow \ p_j \not\leq p_i \tag{22}$$

then the equation (20) may be extended to

$$\mathcal{A} \fatsemi \mathcal{H} = \fatsemi_{p \in P} (\mathcal{A}_p \fatsemi \mathcal{H}_p) = \fatsemi_{p \in P_1} (\mathcal{A}_p \fatsemi \mathcal{H}_p) \fatsemi \ldots \fatsemi_{p \in P_n} (\mathcal{A}_p \fatsemi \mathcal{H}_p) \tag{23}$$

Here, P_1 is the set of code fragments completely analysed at the time of the first break, $P_1 \uplus P_2$ the set completely analysed at the time of the second break, etc.

2.9. Detecting malversion

According to Section 2.7, certification requires that the result $T^\ddagger$ of a computation on the source code C be stored, along with the configuration μ of the analysis process. Above, we have seen that the computation may be carried out in a distributed manner, each distributed part incrementally contributing to the final result. That means formally that the whole computation is a limit

$$T^\ddagger = T^\ddagger_P = \lim_\pi T^\ddagger_\pi \tag{24}$$

for the *directed set* of sets of syntax tree nodes $\pi \subseteq P$ that are closed with respect to pre-dependencies, ordered by inclusion.

That is, P is the full set of syntax tree nodes, and $\pi_1 \leq \pi_2 \;\Leftrightarrow\; \pi_1 \subseteq \pi_2$ for $\pi_1, \pi_2 \subseteq P$, and we are talking about those π for which $\forall p_1, p_2.\ p_1 \leq p_2 \in \pi \;\Rightarrow\; p_1 \in \pi$. These π form a directed set (with limit the full set P) because if π_1, π_2 are closed under pre-dependencies, then so is $\pi_1 \cup \pi_2 \geq \pi_1, \pi_2$.

We say that $T^\ddagger_1 \leq T^\ddagger_2$ for decorated trees $T^\ddagger_1$, $T^\ddagger_2$ if $T^\ddagger_2$ contains at least as much decoration as $T^\ddagger_1$. Recall that 'decoration' is a formula and a bit set (or not) in a field on the syntax tree node record, so $T^\ddagger_1$ and $T^\ddagger_2$ may differ by a bit set in $T^\ddagger_2$ on node p where it is unset in $T^\ddagger_1$ – that is, by the decoration at node p taking a higher value in $T^\ddagger_2$ than it does in $T^\ddagger_1$. We let

$$T^\ddagger_{\{\,\}} = T$$

be the undecorated syntax tree. We let $T^\ddagger_\pi$ be the syntax tree decorated at the points $p \in \pi$ by the analysis and checking procedure. A 'little increment' in the direction defined by the directed set is achieved by moving from π, which contains all the pre-dependencies of p', but not p' itself, to $\pi' = \pi \cup \{p'\}$. That is, $p' \notin \pi$ and $\{p \,|\, p < p'\} \subseteq \pi$. Then the tree decorated at p' as well as at all the $p \in \pi$ is given by applying the analysis and check for node p' to the tree decorated at all the $p \in \pi$:

$$T^\ddagger_{\pi \cup \{p'\}} = T^\ddagger_\pi \,\mathcal{A}_{p'}\,\mathcal{H}_{p'} \tag{25}$$

The intermediate result $T^\ddagger_\pi$ is a partially decorated syntax tree, and $T^\ddagger_{\pi \cup \{p'\}}$ is a slightly more decorated tree. In general, the move from π to $\pi' > \pi$ is given by the following expression:

$$T^\ddagger_{\pi'} = T^\ddagger_\pi \,(\fatsemi_{p \in \pi' - \pi} (\mathcal{A}_p \fatsemi \mathcal{H}_p)) \tag{26}$$

with the order of application on the right governed by the dependency ordering. In particular, starting from zero gives the expression:

$$T^{\ddagger}_{\pi} = T\,(\underset{p\in\pi}{\fatsemi}(\mathcal{A}_p \fatsemi \mathcal{H}_p))$$

and it is the case that proceeding in the direction defined by the directed set increases the amount of decoration:

$$\pi \leq \pi' \;\;\Rightarrow\;\; T^{\ddagger}_{\pi} \leq T^{\ddagger}_{\pi'}$$

so the limit of the $T^{\ddagger}_{\pi}$ is well-formed. It is the 'least' decorated tree that extends all the $T^{\ddagger}_{\pi}$ for $\pi \subseteq P$, the limit being finally achieved at $T^{\ddagger}_{P}$, by the application of an analysis and check at a maximal node p' to the tree $T^{\ddagger}_{P-\{p'\}}$.

The final fully decorated tree $T^{\ddagger} = T^{\ddagger}_{P}$, or indeed any partially decorated approximant $T^{\ddagger}_{P}$, is such that the computation can be checked piecewise. There are inverse computations η_π available to all such that:

$$T^{\ddagger}_{\pi} = T^{\ddagger}\,\eta_\pi \tag{27}$$

The map η_π is simply a 'forgetful' function that drops the decorations on the syntax tree at all nodes p not in π. It works at any later stage $\pi' \geq \pi$:

$$T^{\ddagger}_{\pi} = T^{\ddagger}_{\pi'}\,\eta_\pi \tag{28}$$

If any incorrect intermediate result $\hat{T}^{\ddagger}_{\pi'}$ instead of $T^{\ddagger}_{\pi'}$ were to be introduced into the calculation by some adversary who incorrectly calculates it from $T^{\ddagger}_{\pi}$ for $\pi' = \pi \cup \{p'\}$, leading eventually to an incorrect final result $\hat{T}^{\ddagger}$, it could be detected at any stage $\pi'' \geq \pi'$ via the following calculations:

$$T^{\ddagger}_{\pi} = \hat{T}^{\ddagger}_{\pi''}\,\eta_\pi$$
$$\hat{T}^{\ddagger}_{\pi'} = \hat{T}^{\ddagger}_{\pi''}\,\eta_{\pi'}$$

Then, applying $\mathcal{A}_{p'} \fatsemi \mathcal{H}_{p'}$ to $T^{\ddagger}_{\pi}$, one notes that $\hat{T}^{\ddagger}_{\pi'} \neq T^{\ddagger}_{\pi}\,\mathcal{A}_{p'}\,\mathcal{H}_{p'} = T^{\ddagger}_{\pi'}$. That is what is meant by being able to 'spot-check' the computation. Any attempt at malversion will be detected by any contributer who repeats the lowest corrupted part in the computation hierarchy.

2.10. A category-theoretic characterisation

We present below a characterisation in category-theoretic terms of the properties (set out above) that have allowed the analysis computation to be chopped up, carried out piecewise in a distributed manner, and spot-checked for errors or malversion.

Definition 1 A *distributed process* consists of a directed set D, secondly of a category $\overrightarrow{\mathcal{T}}$ of automorphisms of information of type $\mathcal{T}$ ('decorated syntax trees' in our case, including the undecorated and partially decorated trees) with

automorphisms $f, g : \mathcal{T} \to \mathcal{T}$ as objects and automorphisms $h : \mathcal{T} \to \mathcal{T}$ with $f \fatsemi h = g$ as arrows $h : f \to g$ of $\overrightarrow{\mathcal{T}}$, thirdly of two functors from D to $\overrightarrow{\mathcal{T}}$, a covariant analysis functor $\mathscr{AH}$ and a contravariant 'forgetful' functor $\mathscr{G}$:

$$\begin{array}{ccc} & & D^{\mathrm{op}} \\ & & {\scriptstyle \mathscr{G}}\downarrow \\ D & \xrightarrow{\mathscr{AH}} & \overrightarrow{\mathcal{T}} \end{array} \tag{29}$$

The diagram (29) states that for every $\pi_0, \pi_1, \pi_2 \in D$ with $\pi_0 \leq \pi_1 \leq \pi_2$ there are automorphisms $\mathscr{AH}_{\pi_0}, \mathscr{AH}_{\pi_1}, \mathscr{AH}_{\pi_2}, \mathscr{AH}_{\pi_0,\pi_2}, \mathscr{AH}_{\pi_0,\pi_1}, \mathscr{AH}_{\pi_1,\pi_2} : \mathcal{T} \to \mathcal{T}$ with

$$\begin{aligned} \mathscr{AH}_{\pi_0} \fatsemi \mathscr{AH}_{\pi_0,\pi_1} &= \mathscr{AH}_{\pi_1} \\ \mathscr{AH}_{\pi_1} \fatsemi \mathscr{AH}_{\pi_1,\pi_2} &= \mathscr{AH}_{\pi_2} \\ \mathscr{AH}_{\pi_0,\pi_1} \fatsemi \mathscr{AH}_{\pi_1,\pi_2} &= \mathscr{AH}_{\pi_0,\pi_2} \end{aligned}$$

and the 'forgetful' functor $\mathscr{G}$ satisfies the same scheme in reverse. That is, there are automorphisms $\mathscr{G}_{\pi_0}, \mathscr{G}_{\pi_1}, \mathscr{G}_{\pi_2}, \mathscr{G}_{\pi_0,\pi_2}, \mathscr{G}_{\pi_0,\pi_1}, \mathscr{G}_{\pi_1,\pi_2} : \mathcal{T} \to \mathcal{T}$ such that

$$\begin{aligned} \mathscr{G}_{\pi_1} \fatsemi \mathscr{G}_{\pi_0,\pi_1} &= \mathscr{G}_{\pi_0} \\ \mathscr{G}_{\pi_2} \fatsemi \mathscr{G}_{\pi_1,\pi_2} &= \mathscr{G}_{\pi_1} \\ \mathscr{G}_{\pi_1,\pi_2} \fatsemi \mathscr{G}_{\pi_0,\pi_1} &= \mathscr{G}_{\pi_0,\pi_2} \end{aligned}$$

for $\pi_0, \pi_1, \pi_2 \in D$ with $\pi_0 \leq \pi_1 \leq \pi_2$.

That D is 'directed' means that for each $\pi_1, \pi_2 \in D$, there is $\pi_3 \in D$ with $\pi_1, \pi_2 \leq \pi_3$. In our case:

- D is the *lattice* of dependency-complete sets of syntax tree nodes. Its bottom element $\bot$ is the empty set $\{\}$ and its top element $\top$ is P, the full set of syntax tree nodes. The conjunction $\pi_1 \wedge \pi_2$ in the lattice of two sets $\pi_1,\ \pi_2$ is the set intersection $\pi_1 \cap \pi_2$, and the disjunction $\pi_1 \vee \pi_2$ is the set union $\pi_1 \cup \pi_2$. The lattice D and its opposite D^{op} both have all limits.

- The analysis $\mathscr{AH}_\pi$ is the analysis and check procedure $\fatsemi_{p \in \pi} (\mathcal{A}_p \fatsemi \mathcal{H}_p)$, and the increment $\mathscr{AH}_{\pi_1,\pi_2}$ is the analysis $\fatsemi_{p \in \pi_2 - \pi_1} (\mathcal{A}_p \fatsemi \mathcal{H}_p)$ on the syntax tree nodes in the range $\pi_2 - \pi_1$. It takes as basis the decorations on the syntax tree nodes $p \in \pi_1$ and calculates the decorations on the nodes $p \in \pi_2 - \pi_1$, replacing any existing decorations on those nodes.

- The forgetful map $\mathscr{G}_\pi$ is the function η_π that strips the decoration on trees down to just all nodes $p \in \pi$. The increment $\mathscr{G}_{\pi_1,\pi_2}$ is just the function η_{π_1} applied after η_{π_2} that strips the decoration on $\pi_2 - \pi_1$.

- The whole analysis procedure $\fatsemi_{p \in P} \mathcal{A}_p \fatsemi \mathcal{H}_p$ is the limit $\lim_{\pi \in D} \mathscr{AH}_\pi$.

- The function that strips away all decoration at nodes $p \in P$ is the (co-) limit $\lim_{\pi \in D^{\mathrm{op}}} \mathscr{G}_\pi$.

'Accountability' means that an analysis $\mathscr{AH}_{\pi_2}$ up till point π_2 can be reset to an earlier point $\pi_1 \leq \pi_2$ for spot-checking by forgetting some progress, so that

$$\mathscr{AH}_{\pi_2} \fatsemi \mathscr{G}_{\pi_1,\pi_2} = \mathscr{AH}_{\pi_1}$$

Or, in general for $\pi_1 \leq \pi_2, \pi_3 \leq \pi_4$, one may partially reverse an analysis increment and then reapply the discarded part and more:

$$\mathscr{AH}_{\pi_2} \fatsemi \mathscr{G}_{\pi_1,\pi_2} \fatsemi \mathscr{AH}_{\pi_1,\pi_3} = \mathscr{AH}_{\pi_2} \fatsemi \mathscr{AH}_{\pi_2,\pi_4} \fatsemi \mathscr{G}_{\pi_3,\pi_4}$$

which means that the following diagram commutes:

$$\begin{array}{ccc} \mathscr{AH}_{\pi_2} & \xrightarrow{\mathscr{AH}_{\pi_2,\pi_4}} & \mathscr{AH}_{\pi_4} \\ {\scriptstyle \mathscr{G}_{\pi_1,\pi_2}}\downarrow & & {\scriptstyle \mathscr{G}_{\pi_3,\pi_4}}\downarrow \\ \mathscr{AH}_{\pi_1} & \xrightarrow{\mathscr{AH}_{\pi_1,\pi_3}} & \mathscr{AH}_{\pi_3} \end{array} \tag{30}$$

Definition 2 An *accountable distributed process* is a distributed process that satisfies (30) for all $\pi_1 \leq \pi_2, \pi_3 \leq \pi_4$ in D.

We can now provide a category theoretic statement of what it means for $\mathscr{AH}$ to be an accountable distributed process. For background on the concepts of category theory used below, the reader is referred to [ML98].

Let E be the suborder of $D \times D$ defined by requiring the pair of left hand corners of diagram (30) to lie below the pair of right hand corners in the ordering; that is $(\pi_1, \pi_2) \leq (\pi_3, \pi_4) \;\; \Leftrightarrow \;\; \pi_1 \leq \{\pi_2, \pi_3\} \leq \pi_4$.

One may understand the diagram as exhibiting an arrow $(\pi_1, \pi_2) \leq (\pi_3, \pi_4)$ in E as an arrow from object $(\pi_2, \mathscr{G}_{\pi_1,\pi_2}, \pi_1)$ on the left in the diagram to object $(\pi_4, \mathscr{G}_{\pi_3,\pi_4}, \pi_3)$ on the right in the *comma category* $(\mathscr{AH} \downarrow \mathscr{AH})$. Moreover, that choice is functorial; the diagrams combine associatively. This functor $G : E \to (\mathscr{AH} \downarrow \mathscr{AH})$ formally takes the arrow $(\pi_1, \pi_2) \leq (\pi_3, \pi_4)$ of E to the arrow $(\pi_2 \leq \pi_4, \pi_1 \leq \pi_3)$ of $(\mathscr{AH} \downarrow \mathscr{AH})$, the images $\mathscr{AH}_{\pi_2,\pi_4}$, $\mathscr{AH}_{\pi_1,\pi_3}$ of which form top and bottom of the diagram, while source and target objects set the sides.

Let $F : (\mathscr{AH} \downarrow \mathscr{AH}) \to D \times D$ be the functor that takes a source object $(\pi_2, f : \mathscr{AH}_{\pi_2} \to \mathscr{AH}_{\pi_1}, \pi_1)$ of $(\mathscr{AH} \downarrow \mathscr{AH})$ to the target object (π_1, π_2) of $D \times D$, and maps an arrow $(\pi_2 \leq \pi_4, \pi_1 \leq \pi_3)$ to the relation $(\pi_1, \pi_2) \leq (\pi_3, \pi_4)$ between objects of $D \times D$. Thus F forgets labels on the arrows inside object triples, mapping one ordering, $(\mathscr{AH} \downarrow \mathscr{AH})$, with many nodes into another, $D \times D$, with fewer nodes. The relation produced as an image by F in principle may be all of $D \times D$, but the image produced by the composite $G \fatsemi F$ is just exactly E. That is

$$I_E = G \fatsemi F$$

where I_E is the embedding of E in $D \times D$. Any functor $G : E \to (\mathscr{AH} \downarrow \mathscr{AH})$ that satisfies this equation supplies the functor $\mathscr{G}$, by defining for $\pi_1 \leq \pi_2$ the arrow $\mathscr{G}_{\pi_1,\pi_2}$ as being that given by $(\pi_2, \mathscr{G}_{\pi_1,\pi_2}, \pi_1) = G(\pi_1, \pi_2)$. Thus:

Proposition 1 *An* accountable distributed process $\mathscr{AH} : D \to \overrightarrow{\mathcal{T}}$ *is one for which the functor* $F : (\mathscr{AH} \downarrow \mathscr{AH}) \to D \times D$ *that drops the arrow component from each triple comprising a comma category object has a pre-inverse* G *with* $I_E = G \fatsemi F$*, where* E *is the suborder of* $D \times D$ *defined by* $(\pi_1, \pi_2) \leq (\pi_3, \pi_4) \;\Leftrightarrow\; \pi_1 \leq \{\pi_2, \pi_3\} \leq \pi_4$.

The condition of the proposition says there is a G making the diagram below commute:

$$\begin{array}{ccc} E & \xrightarrow{\;I\;} & E \\ {\scriptstyle G}\downarrow & & \downarrow{\scriptstyle I_E} \\ (\mathscr{AH} \downarrow \mathscr{AH}) & \xrightarrow{\;F\;} & D \times D \end{array}$$

Applied to some clean starting point $T = (\lim_D \mathscr{G})(T)$, the analysis produces a limit $T^\ddagger = (\lim_D \mathscr{AH})(T)$ that can be achieved incrementally, and the final result provides accountability by virtue of being checkable: it can be returned to an earlier stage of analysis via $\mathscr{AH}_\pi(T) = \mathscr{G}_\pi(T^\ddagger)$.

3. Logic

This section provides a brief and self-contained account of the programming logic of imperative languages that forms the basis of the formal method in our system. A semantic model for programs as sets of traces is set out in Appendix A, where the logic is shown to be sound with respect to it.

The program logic generates the assertions decorating the syntax tree $T^\dagger$. It is called NRBG, for 'normal, return, break, goto', the kinds of program flow treated. The logic in this form was first published in [BG04] but has its origins in a '3-phase' ('begin, during, end') logic [BMSD95, BMS$^+$96] developed for specifying and reasoning about hardware behaviour. The driving idea is that a program contains a *normal* flow of control – which passes from the beginning of each statement through its end to the beginning of the next in sequence and which passes at conditional statements through one of the two branches according to the outcome of the test – and also *abnormal* flows of control. The latter flows exit a piece of code before what we think of as the end is reached, 'during' the course of execution. For example, a break statement causes an early exit from the body of a loop (via a 'break flow'), without the statements following it in normal sequence in the loop body ever being executed. Independently, Henrik Tews [Tew04] also formalised the distinct flows in an imperative program, but developed the idea in the direction of a denotational semantics that does not use the artifice of continuation programming as foundation for goto semantics, with less emphasis on programming logic.

The thinking is best illustrated with reference to sequential statements. Suppose the condition p holds initially, and consider what happens when the code sequence $a; b$ runs. Either it terminates normally with condition r holding,

or it terminates abnormally with condition x holding. That is, the following assertion-code-assertion triple (just 'triple', from now on) holds:

$$\{p\}\ a; b\ \{\mathbf{N}r \vee \mathcal{E}x\}$$

where N stands for 'normal' and $\mathcal{E}$ stands for any of the $\mathbf{R}$ ('return'), $\mathbf{B}$ ('break'), $\mathbf{G}_l$ ('goto') exceptional flows.

This situation may ensue in one of two ways. The first is that the code fragment a terminates normally with condition q holding and fragment b continues from the state in which q holds to the asserted termination conditions. That is, both of the following triples hold:

$$\{p\}\ a\ \{\mathbf{N}q\} \text{ and } \{q\}\ b\ \{\mathbf{N}r \vee \mathcal{E}x\}$$

The second possibility is that the fragment a terminates abnormally with condition x before b has a chance to run. That is, the following triple holds:

$$\{p\}\ a\ \{\mathcal{E}x\}$$

Two rules of deduction correspond to these two judgements. They are:

$$\frac{\{p\}\ a\ \{\mathcal{E}x\}}{\{p\}\ a; b\ \{\mathcal{E}x\}}[\text{seq(a)}] \quad \frac{\{p\}\ a\ \{\mathbf{N}q\} \quad \{q\}\ b\ \{\mathbf{N}r \vee \mathcal{E}x\}}{\{p\}\ a; b\ \{\mathbf{N}r \vee \mathcal{E}x\}}[\text{seq(b)}]$$

The two, **seq(a)** and **seq(b)**, can be combined into one single **seq** rule:

$$\frac{\{p\}\ a\ \{\mathbf{N}q \vee \mathcal{E}x\} \quad \{q\}\ b\ \{\mathbf{N}r \vee \mathcal{E}x\}}{\{p\}\ a; b\ \{\mathbf{N}r \vee \mathcal{E}x\}}[\text{seq}]$$

See [BG04, BP06b] for fuller justifications of the logic. Early presentations used a set of different but interacting logics $\mathbf{N}$, $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}$ and the presentation here (Table 1) is innovative in that it instead introduces the $\mathbf{N}$, $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_l$ as modal operators . The advantage of doing things this way is that the number of logical rules falls to about ten, from about twenty. The modal operators distribute over logical disjunction and conjunction, and are orthogonal, idempotent and flat (i.e. preserve falsity). See Table A5 in the Appendix here for a formalisation of the algebra of the modal operators.

The logic is compositional, including that for **goto** statements. They are handled by hypothesising a 'programming contract' p_l at label l in the code. If the condition p_l is loose enough that it covers all the states subsequently deduced to hold at **goto** l statements, then the contract is valid. If p_l is set too broad initially, then that is not wrong, but it may be too weak to prove a desired result with. In practice, p_l is set to false initially and relaxed during the course of reasoning, following the 'proof procedure' set out in Remark AR5, until a just sufficiently broad contract condition p_l for each label l is found.

The reasoning for loops can be boiled down to reasoning about a 'forever loop', a loop with an exit condition that can never trigger. Breaking from the body with condition q is exiting the loop normally with condition q. I.e.:

$$\frac{\{p\}\ a\ \{\mathbf{B}q \vee \mathbf{N}p\}}{\{p\}\ \mathbf{while}(\mathbf{true})\ a\ \{\mathbf{N}q\}}[\text{whl(a)}]$$

Table 1: Deduction rules of NRBG program logic. For the **seq** and **if** rules, $\mathcal{E}$ is any of $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_l$; for **whl**, $\mathcal{E}$ is any of $\mathbf{R}$, $\mathbf{G}_l$; for **lbl**, $\mathcal{E}$ is any of $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_{l'}$ with $l' \neq l$. In the **lbl** rule, label l must not appear anywhere outside of a.

$$\frac{\triangleright\ \{p\}\ a\ \{\mathbf{N}q \vee \mathcal{E}x\} \quad \triangleright\ \{q\}\ b\ \{\mathbf{N}r \vee \mathcal{E}x\}}{\triangleright\ \{p\}\ a;b\ \{\mathbf{N}r \vee \mathcal{E}x\}}[\text{seq}] \qquad \frac{\triangleright\ \{p\}\ a\ \{\mathbf{B}q \vee \mathbf{N}p \vee \mathcal{E}x\}}{\triangleright\ \{p\}\ \mathbf{while}(\mathbf{true})\ a\ \{\mathbf{N}q \vee \mathcal{E}x\}}[\text{whl}]$$

$$\frac{}{\triangleright\ \{p\}\ \mathbf{return}\ \{\mathbf{R}p\}}[\text{ret}]$$

$$\frac{}{\triangleright\ \{p\}\ \mathbf{break}\ \{\mathbf{B}p\}}[\text{brk}] \qquad [p \to p_l]\frac{}{\mathbf{G}_l\, p_l \triangleright\ \{p\}\ \mathbf{goto}\ l\ \{\mathbf{G}_l\, p\}}[\text{go}]$$

$$\frac{\triangleright\ \{p \wedge c\}\ a\ \{\mathbf{N}q \vee \mathcal{E}x\} \quad \triangleright\ \{p \wedge \neg c\}\ b\ \{\mathbf{N}q \vee \mathcal{E}x\}}{\triangleright\ \{p\}\ \mathbf{if}(c)\ a\ \mathbf{else}\ b\ \{\mathbf{N}q \vee \mathcal{E}x\}}[\text{if}] \qquad \frac{}{\triangleright\ \{q[e/x]\}\ x{=}e\ \{\mathbf{N}q\}}[\text{let}]$$

$$\frac{\mathbf{G}_l\, p_l \triangleright\ \{p_l\}\ b\ \{q\} \quad \mathbf{G}_l\, p_l \triangleright\ \{p\}\ a;b\ \{q\}}{\mathbf{G}_l\, p_l \triangleright\ \{p\}\ a;l{:}b\ \{q\}}[\text{frm}] \qquad \frac{\mathbf{G}_l\, p_l \triangleright\ \{p\}\ a\ \{\mathbf{G}_l p_l \vee \mathbf{N}q \vee \mathcal{E}x\}}{\triangleright\ \{p\}\ a\ \{\mathbf{N}q \vee \mathcal{E}x\}}[\text{lbl}]$$

On the other hand, leaving the body of the loop with an abnormal flow different than a break flow causes the loop to terminate with that abnormal condition:

$$\frac{\{p\}\ a\ \{\mathcal{E}x \vee \mathbf{N}p\}}{\{p\}\ \mathbf{while}(\mathbf{true})\ a\ \{\mathcal{E}x\}}[\text{whl(b)}]$$

Here $\mathcal{E}$ stands for any of $\mathbf{R}$, $\mathbf{G}_l$, where l is not a label declared in a. The two, **whl(a)** and **whl(b)**, can be combined into one single **whl** rule:

$$\frac{\{p\}\ a\ \{\mathcal{E}x \vee \mathbf{B}q \vee \mathbf{N}p\}}{\{p\}\ \mathbf{while}(\mathbf{true})\ a\ \{\mathbf{N}q \vee \mathcal{E}x\}}[\text{whl}]$$

A finite loop can be implemented as a forever loop with a conditional **break** at the head of the body.

As indicated above, the rules dealing with **goto** statements are slightly more delicate. Hypotheses p_l for the 'contracted' sets of states that may occur at labels l appear as the p_l in the $G_l\, p_l$ left of the '$\triangleright$' sign in the rules. Generally, contracts pass unchanged from top to bottom of the rule, and we have elided them where that is the case in Table 1, which shows ten rules of NRBG logic corresponding one-for-one to the ten imperative constructs of C shown.

A **goto** l does nothing but exit in a $\mathbf{G}_l$ flow, provided it satisfies the contract:

$$\frac{}{\mathbf{G}_l\, p_l \triangleright\ \{p_l\}\ \mathbf{goto}\ l\ \{\mathbf{G}_l\, p_l\}}[\text{go}]$$

The contracts $\mathbf{G}_l\, p_l$ saying what happens at **goto** l statements must cover the flow out via this particular **goto** l statement, but the entry condition may be stricter. The most general form of the rule is guarded as follows:

$$[p \to p_l]\frac{}{\mathbf{G}_l\, p_l \triangleright\ \{p\}\ \mathbf{goto}\ l\ \{\mathbf{G}_l\, p\}}[\text{go}]$$

Contracts $\mathbf{G}_l\, p_l$ on the left of the '$\triangleright$' also get used when reasoning about labelled statements. A label is an alternative point of entry into the program, so if the

hypothesis is that the states at **goto** l statements are covered by p_l, then in order to be assured of moving from an initial state described by p through a labelled fragment of code $l : b$ to a postcondition q, one has to take into account that the initial states for the fragment b are described by the disjunction $p \vee p_l$, because of the possible incoming **goto** flows to the label. That is:

$$\frac{\mathbf{G}_l\, p_l \ \triangleright\ \{p \vee p_l\}\ b\ \{q\}}{\mathbf{G}_l\, p_l \ \triangleright\ \{p\}\ l : b\ \{q\}}$$

The table shows the most general form of this rule. Contract $\mathbf{G}_l\, p_l$ can only be discharged (rule **lbl** in the table) when there remain no references to the label l outside of the piece of code a under consideration.

We have not listed the logic for subroutine calls because analysis always inlines the code or treats the call as opaque satisfying assertions to be tested. Table A6 in Appendix A gives the full semantics and logic of calls. It describes an underlying abstract language C from which have been derived the rules of Table 1 for the concrete language C.

4. Implementation

This section briefly recapitulates an experiment [BP09] repeating the analysis previously carried out using a monolithic toolset [BG04, BP06b], using the 'volunteer cloud' platform instead, enabling the performances to be compared. The account here makes concrete the account of the analysis and checking process presented in Section 2.7, laying bare the statistics of the procedure.

4.1. Experiment

The aim of the experiment was to prove that a large formal verification problem could be solved via an ad-hoc distributed network of automated solvers.

The target code consisted of the Linux kernel source code (written in assembler and C [Ame89, Int99]) and the analysis objective was to detect a particular kind of runtime deadlock known as 'sleep under spinlock' in the operating system as compiled for multi-CPU 32-bit Intel (IA32) platforms. Those faults detected during the experiment are not intrinsically specific to the Intel platform, however, because 80-90% of the code is shared with and common to the 15 other major architectural types supported by the Linux kernel, and any faults found in a common section are relevant to the other platforms too. The experiment detected about three such faults per million lines of code. It also simultaneously checked for other similar deadlock possibilities (notably 'spinlock under spinlock under spinlock ... '), which are detected at close to the same frequency. The average lifetime from appearance to elimination in the source code of the faults detected appears to be about six months, checking against the version histories.

4.2. Architecture and procedure

The platform in this experiment comprised a set of solver clients plus a single task server, a Linux host running the postgresql [Dou05] DBMS. The task server split the analysis task into smaller subtasks in accord with the theory presented in Section 2 and distributed the subtasks to the remote clients.

4.3. Uploading the verification problem to the task server

In order to make use of the volunteer network, a problem submitter first parses the source code to be analysed then uploads the resulting syntax tree to the remote database. However, for a code base the size of the Linux kernel (ten million lines of code), populating the remote database with the syntax tree turns out to be a logistical problem too difficult to treat naively.

In the experiment [BP09], only a million or so lines of Linux kernel source code was offered for analysis, corresponding to an average set of compilation configuration choices. Nevertheless, that amount of source code gave rise to over ten million syntax tree nodes. Each insertion involved several relational database updates on the postgresql back-end and the acknowledgement and locking requirements slowed the transactions down to as much as a second or more across the network (the average time was a tenth of a second or so). Thus, with a naive approach, simply uploading the problem for analysis would have taken on the order of days or weeks, a completely unrealistic scenario.

This 'database population problem' was eventually solved by first writing the parse data to a fast local non-relational data store (a GNU DBM 1.8.3 -based store), then copying it to the remote database site in one chunk, converting it to relational database format locally on the database server. Using this procedure, the database was populated in a single day. In a production scenario, the source code would have to be uploaded whole to a separate service that can populate the database from close by.

The call graph was extracted from the parse and used to compute the list of functions that 'sleep' (can be swapped out of the kernel), for later use. Functions 'sleep' if they call a function that sleeps. There is a set of 50 known functions that sleep and make no calls to other functions, from which the rest are derived.

4.4. Reducing the number of subtasks

In practice, a single task downloaded for solution by a volunteer client consists of the analysis of a single top-level functional unit. However, in our experiment it turned out that many of the function definitions from common header files had effectively been duplicated tens, hundreds and even thousands of times through being declared *static* and *inline*. In C, this combination signals local scope and context at every implantation site. See Fig. 1 for a count of the number of implanted definitions; on the right of the figure the dozens of function definitions implanted into more than a thousand different sites can be observed. The number of analysis tasks was reduced tenfold by analysing only one representative from each class of syntactically identical functional definitions.

The assumption is that no two syntactically identical definitions captured identically named but different external references, which holds good for well-written code. Out of three quarters of a million top-level function definitions in the database, only seventy-two thousand corresponded to non-duplicates.

4.5. Optimisation

Having the client solvers fetch data from the database server as needed turned out to be too inefficient as a general strategy. The latency of each

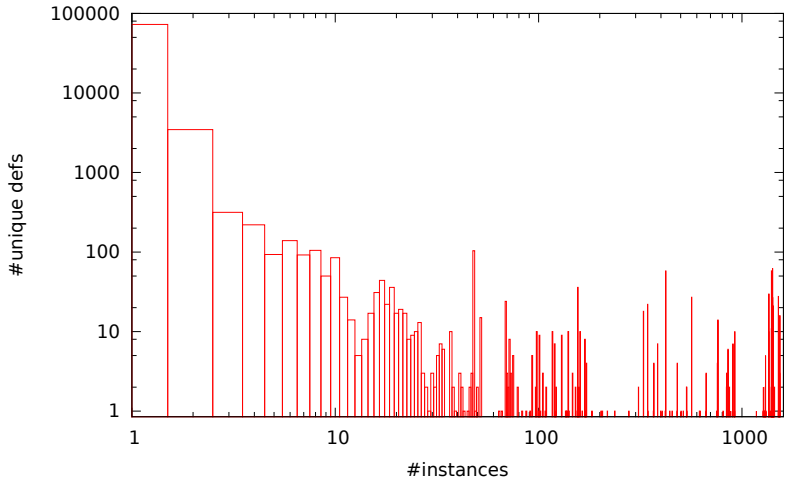

Figure 1: Top-level definitions with multiple instances ($\sum xy = 746844$). At extreme left are near 100000 unique definitions, at extreme right are near 100 definitions copied through the code base as separate instances thousands of times each.

transaction was such that the computation proceeded about one thousand times as slowly as it would have done on locally stored data (using the experiment in [BP06b] for comparison). We looked for performance optimisations:

1. *Avoid clients downloading syntax trees*: this ensured that volunteer clients did not download syntax trees node by node but instead downloaded corresponding source text, re-parsing it locally.
2. *Use client-side caching*: a persistent cache was added on the client side just atop the database interface. The hit-rate for the cache was around 95% with corresponding performance increase (i.e., 20 times).
3. *Pre-compute for database-intensive tasks*: the few database interactions that turned out to take minutes each – queries involving complex searches and aggregates across millions of database entries, such as calculating new priorities for the remaining work tasks after each task completion by a volunteer client – were amortised by calculating up to five hundred results ahead of time and then doling them out as needed.
4. *Use theorem-proving to reduce task complexity*: significant reductions in the complexity of the logical formulae generated during the processing were achieved by incorporating automatic theorem-proving techniques into the mechanisms that generate the formulae and/or by using abstract interpretation before performing the mixed integer linear programming -based analysis (See Section 2.5 and [BP09]).

Further improvements were achieved through task selection, as explained below.

4.6. Allocating tasks to clients

The strategy for allocating work to volunteer clients can significantly affect performance. The group of volunteer clients makes more progress overall if they complete the easy work tasks first. However, there is no way of knowing *a priori* which tasks are easy: the only sure way of finding out is by executing them.

The size of analysis task taken on by volunteer clients was initially set to 'one complete functional unit', i.e. a top-level function definition, comprising a single connected 'tree' from the abstract syntax 'forest' in the database. Each functional unit was initially assumed to be equally as hard to analyse as every other. Each volunteer was initially given $T_0 = 10$ minutes of CPU time (normalised to a 1GHz CPU) in which to complete the work task. If the limit was exceeded, the client abandoned the task, reported back the incompletion statistic to the database, and moved on to a different work task. The task's estimate of intrinsic difficulty was raised, reflected in an increased timeout value $T_1 > T_0$.

This tamps down as much as possible on concurrent interactions with the database server. Giving volunteer clients by default a relatively large work unit to process reduces the number of data requests transmitted across the network and thus in principle helps the computation overall. The downside is that clients may be given more than they can deal with, plugging progress overall. Imposing a timeout per task was the simplest cure though it implied the loss of the data accumulated by the client up to the point of abandonment.

1. *Hard tasks.* Every time a work task was abandoned uncompleted, the estimated time required to complete it was increased by 50% (i.e., $T_{n+1} = 1.5\,T_n$), so that the next client to take it on would spend longer on it before abandoning. Tasks with a higher timeout were handed out with lower frequency (i.e., with lower priority) so that clients would tend to take the easier tasks first. Abandonment wastes the earlier effort put in, but the time taken overall is still dominated by the successful final stint. Those 'hard' work tasks that took longer than the initial 10 minutes turned out to comprise only 0.5% (three hundred-odd) of the total number.
2. *Very hard tasks.* The tasks that took longer than an hour without completion (a hundred or so, or 0.15% of the original total) were dealt with syntax tree node by syntax tree node in accord with the theory developed in Section 2. The incremental progress in the client dealing with them was check-pointed to the database every minute, pushing up the number of remote database transactions in return for guaranteed progress. Any volunteer client could take up the work where another left off.
   That eventually successfully dealt with all but 0.03% of the original set of seventy-two thousand functional units submitted for analysis.
3. *Ultra hard tasks.* Twenty or so function definitions remained intractable. A few contained constructs peculiar to GNU C that could not be handled by the parser, 'interior' (local) function definitions within other function definitions being the majority contributor. The rest were characterised by the presence of generated symbolic logical assertions of great complexity, containing more than 40,000 terms each.

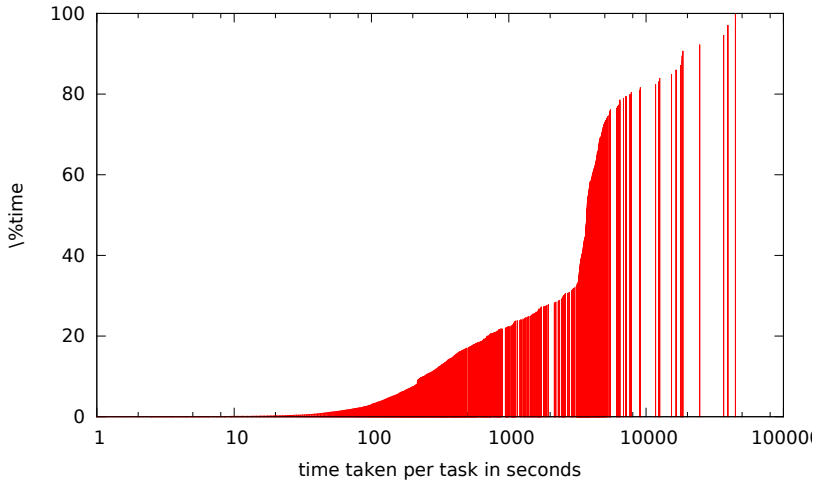

Figure 2: Percentage of total time taken per analysis task (cumulative).

Fig. 2 plots the time taken per task against percentage of the total time taken.

The graph shows that the tasks taking up to an hour of computation time made up only about 30% of the total time. There is a clear inflexion point at about 3600s (the point at which tasks were shifted to checkpointing execution). Two thirds of the computation time overall was spent on those only a hundred or so 'very hard' tasks that in number comprised only 0.15% of the total numbers. In itself, that is very surprising.

4.7. Inferences from the statistics

It should take around 500 1GHz volunteer clients to complete the work undertaken in the experiment in under six hours, under the same conditions (which, however, were nowhere near optimal for performance). A rough average time needed overall for processing per top-level functional unit in the source code was 116 seconds on a notional 1GHz CPU. The 'very hard tasks' (taking more than an hour), though they accounted for not much more than 0.15% of the numbers, required around 70% of the processing time. See Fig. 3 for a straightforward graph of the timing spreads.

The CPU load on a volunteer was rarely more than a few percent for 99.5% of the tasks undertaken, rising towards maximum only on the 'hard' tasks. The implication is that the clients were generally I/O bound, or CPU load would have been much higher.

It is notable that, compared to the original monolithic implementation, the networked computation took 'about 50 times as long' per client, making parity with respect to the original experiment at about the 50-client mark. But the original computation threw away its intermediate calculations, meaning that accountability would have required repeating the whole computation from scratch

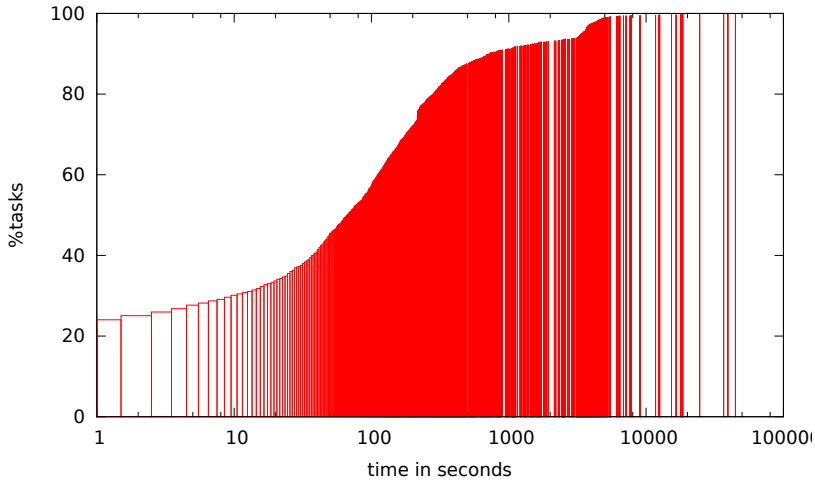

Figure 3: Time taken per analysis task (cumulative count).

– it was not a scalable solution, while the volunteer network approach is scalable. The bottleneck is believed to have been the database server, which did not have quite enough memory to hold the whole database at one time.

The network on which the server sat was not close to limiting (running at no more than a few percent of the 100Mb/s capacity), but the ADSL router behind which it sat was inherently asymmetric, with the outward half limited to no more than 20% of the 10Mb/s inward half. Nevertheless, those networking limits were not reached either. Database queries and responses did not exceed more than a hundred or so bytes each, and no more than a few hundred per second requests ever came in. It was the response times to queries at the server itself that dominated, hence the conclusion that the database server limited. The indications are that there was not quite enough memory (3GB) in the server to fit the whole of the database in RAM, with consequent swapping to disk, and network throughput was not limiting.

We can estimate the quantitative effect of lifting our server's limits: assuming each client possibly could run as fast as in our original monolithic experiment, then each would emit 50 times as many queries per second as in our distributed experiment, and n clients would run the distributed experiment n times as fast as the monolithic experiment. But $50n$ times as many queries arriving at the server per second is about $5000n$ queries per second, or about $500000n$ bytes per second across the network. If 5 MB/s total per server sustained across a wide-area network is realistic, then only 10 clients could operate! It would be more practical to throttle each client to 10% and run 100 clients, for a speed-up of 10 times over a monolithic configuration.

5. Summary

The computation of a certificate guaranteeing the absence of formally defined defects in an open source code base has been described in formal terms.

It has been shown that the computation may be handled incrementally by a distributed 'volunteer cloud' of client CPUs, where we use the term to describe an amorphous network of computers belonging to anonymous volunteers in which each volunteer can also be a problem submitter, each taking a fragment of the work upon themselves at a time. This structure in the computation allows it to be rechecked one piece at a time without starting from the beginning each time, such that errors or deliberate malfeasance will be discovered. This property has also been characterised in more general category-theoretic terms.

An experiment in which the 'volunteer cloud' was organised to analyse about a million lines of C code (requiring about nine million seconds of standardised 1GHz CPU time) has validated the ideas here. Although individual tasks were solved slowly compared to dedicated local hardware, that is not a real issue, firstly because the volunteers who benefit from and contribute to this kind of cloud do not want to host expensive hardware, or run it all the time, secondly because enough volunteers do get the job more quickly, thirdly because distributing the computation mutiple times over is crucial to accountability, and fourthly because the bottleneck in our experiment turned out to be our database server, implying that raising the ceiling on server performance (as simple as 'more memory so the database can be held in RAM', or 'multiple servers') results in a direct improvement for the whole system.

References

[ABK09] F. Abujarad, B. Bonakdarpour, S. Kulkarni. Parallelizing Deadlock Resolution in Symbolic Synthesis of Distributed Programs. In *Proc. PDMC 2009: 8th International Workshop on Parallel and Distributed Methods in Verification*. Nov. 2009.

[AKW05] D. P. Anderson, E. Korpela, R. Walton. High-Performance Task Distribution for Volunteer Computing. In *Proc. First IEEE International Conference on e-Science and Grid Technologies*. Dec. 2005.

[Ame89] American National Standards Institute. American National Standard for Information Systems – Programming Language C, ANSI X3.159-1989. 1989.

[And04] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proc. 5th IEEE/ACM International Workshop on Grid Computing*. Nov. 2004.

[BB12] P. Breuer, J. Bowen. Typed Assembler for a RISC Crypto-Processor. In Barthe et al. (eds.), *Proc. International Symposium on Engineering Secure Software and Systems (ESSoS'12)*. LNCS 7159, pp. 22–29. Springer, Feb. 2012.

[BG04] P. T. Breuer, M. García Valls. Static Deadlock Detection in the Linux Kernel. In Llamosí and Strohmeier (eds.), *Proc. 9th Ada-Europe International Conference on Reliable Software Technologies*. LNCS 3063, pp. 52–64. Springer, June 2004.

[BKS04] M. Barnett, M. L. K. Rustan, W. Schulte. The Spec# Programming System: An Overview. In *Proc. CASSIS 2004*. LNCS 3362. Springer, 2004.

[BMS+96] P. Breuer, N. Martinez Madrid, L. Sanchez, A. Marín, C. Delgado Kloos. A formal method for specification and refinement of real-time systems. In *Real-Time Systems, Proceedings of the Eighth Euromicro Workshop on*. Pp. 200–204. 1996.
doi: 10.1109/EMWRTS.1996.557891

[BMSD95] P. Breuer, N. Martinez Madrid, L. Sanchez, C. Delgado Kloos. An algebra for VHDL with signal attributes. In *Asia-Pacific Conference on Hardware Definition Languages*. Pp. 99–106. 1995.

[Bob01] P. K. Bobko. Open-Source Software and The Demise Of Copyright. *Rutgers Computer & Technology Law Journal* 51(1), 2001.

[BP06a] P. T. Breuer, S. J. Pickin. Checking for Deadlock, Double-Free and Other Abuses in the Linux Kernel Source Code. In *Proc. Workshop on Computational Science in Software Engineering (CSSE'06), co-located with International Conference on Computational Science (ICCS 2006)*. LNCS 3994, pp. 765–772. Springer, May 2006.

[BP06b] P. T. Breuer, S. J. Pickin. One Million (LOC) and Counting: Static Analysis for Errors and Vulnerabilities in the Linux Kernel Source Code. In Pinho and Harbour (eds.), *Proc. 11th Ada-Europe International Conference on Reliable Software Technologies*. LNCS 4006, pp. 56–70. Springer, June 2006.

[BP06c] P. T. Breuer, S. J. Pickin. Symbolic Approximation: An Approach to Verification in the Large. *Innovations in Systems and Software Engineering* 2(3):147–163, Oct. 2006.

[BP06d] P. T. Breuer, S. J. Pickin. Verification in the Large via Symbolic Approximation. In *Proc. Leveraging Applications of Formal Methods, Second International Symposium (ISoLA 2006)*. Pp. 408–415. IEEE Conference Publications, Nov. 2006.
doi: 10.1109/ISoLA.2006.1

[BP07] P. Breuer, S. Pickin. Verification in the Light and Large: Large-Scale Verification for Fast-Moving Open Source C Projects. In *Software Engineering Workshop 2007 (SEW 2007), 31st IEEE*. Pp. 246–255. 2007.
doi: 10.1109/SEW.2007.37

[BP09] P. T. Breuer, S. Pickin. A Formal Nethod (A Networked Formal Method). *Innovations in Systems and Software Engineering*, Dec. 2009.
doi: 10.1007/s11334-009-0121-4

[BPL06] P. T. Breuer, S. J. Pickin, M. Larrondo Petrie. Detecting Deadlock, Double-Free and Other Abuses in a Million Lines of Linux Kernel Source. In *Proc. Software Engineering Workshop 2006 (SEW'06), 30th Annual IEEE/NASA*. Pp. 223–233. IEEE Press, New York, Apr. 2006.
doi: 10.1109/SEW.2006.15

[BR02] T. Ball, S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proc. POPL'02: ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*. 2002.

[CC77] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM Symposium on the Principles of Programming Languages*. Pp. 238–252. 1977.

[CDPS09] V. Cunsolo, S. Distefano, A. Puliafito, M. Scarpa. Cloud@Home: Bridging the Gap between Volunteer and Cloud Computing. In Huang et al. (eds.), *Emerging Intelligent Computing Technology and Applications*. LNCS 5754, pp. 423–432. Springer, Berlin/Heidelberg, 2009.
doi: 10.1007/978-3-642-04070-2_48

[CES86] E. Clarke, E. Emerson, A. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8(2):244–253, 1986.

[Dou05] K. Douglas. *PostgreSQL*. Sams Publishing, 2 edition, 2005.

[ECCH00] D. Engler, B. Chelf, A. Chou, S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proc. 4th Symposium on Operating System Design and Implementation (OSDI 2000)*. Pp. 1–16. Oct. 2000.

[EL02] D. Evans, D. Larochelle. Improving Security using Extensible Lightweight Static Analysis. *IEEE Software*, Jan/Feb 2002.

[FTA02] J. S. Foster, T. Terauchi, A. Aiken. Flow-Sensitive Type Qualifiers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002)*. Pp. 1–12. June 2002.

[GH] J. V. Guttag, J. J. Horning. Introduction to LCL, A Larch/C Interface Language. http://ftp.digital.com/pub/Compaq/SRC/research-reports/abstracts/src-rr-074.html.

[Gom99] R. W. Gomulkiewicz. How Copyleft Uses License Rights to Succeed in The Open Source Software Revolution and The Implications for Article 2B. *Houston Law Review* 179(36), 1999.

[Gri02] A. Griffith. *GCC: The Complete Reference*. McGrawHill/Osborne, 2002.

[HJG08a] G. J. Holzmann, R. Joshil, A. Groce. Model Driven Code Checking. *Automated Software Engineering* 15(3-4), Dec. 2008.

[HJG08b] G. J. Holzmann, R. Joshil, A. Groce. Swarm Verification. In *Proc. ASE 2008, 23rd IEEE/ACM International Conference on Automated Software Engineering*. Sept. 2008.

[HKA12] E. M. Heien, D. Kondo, D. P. Anderson. A Correlated Resource Model of Internet End Hosts. *IEEE Transactions on Parallel and Distributed Systems* 23:977–984, 2012.
doi: 10.1109/TPDS.2011.251

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM* 21(8):666–677, Aug. 1978.
doi: 10.1145/359576.359585

[Hol03] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Sept. 2003.

[Int99] International Standards Organisation. ISO/IEC 9899-1999, Programming Languages - C. 1999.

[JW04] R. Johnson, D. Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proc. 13th USENIX Security Symposium*. Aug. 2004.

[LKM12] D. Lázaro, D. Kondo, J. M. Marqués. Long-term Availability Prediction for Groups of Volunteer Resources. *Journal of Parallel and Distributed Computing* 72(2):281–296, 2012.
doi: 10.1016/j.jpdc.2011.10.007
`http://www.sciencedirect.com/science/article/pii/S0743731511002061`

[ML98] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. Springer, 1998.

[NL96] G. C. Necula, P. Lee. Safe Kernel Extensions without Run-Time Checking. *SIGOPS Operating Systems Review* 30:229–243, Oct. 1996.

[Ray01] E. S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Chapter 5: The Magic Cauldron. O'Reilly, Feb. 2001.
doi: 10.1.1.37.6511

[SBA+10] B. Segal, P. Buncic, C. Aguado Sanchez, J. Blomer, D. García Quintas, A. Harutyunyan, P. Mato, J. Rantala, D. Weir, Y. Yao. LHC Cloud Computing with CernVM. In *Proc. 13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research.* Proceedings of Science, Feb. 2010. `http://pos.sissa.it/archive/conferences/093/004/ACAT2010_004.pdf`

[Sch98] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1998.

[Tew04] H. Tews. Verifying Duff's device: A simple compositional denotational semantics for Goto and computed jumps. Technical report, Institut f ur Theoretische Informatik, TU Dresden, 2004. Draft.

[WFBA00] D. Wagner, J. S. Foster, E. A. Brewer, A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. Network and Distributed System Security (NDSS) Symposium 2000.* Feb. 2000.

A. Rationale for NRBG logic

This section gives a *semantic model* for an extension NRBG(E) of the NRBG logic in Table 1. The extra 'E' part treats of exceptions (catch/throw in Java, setjmp/longjmp in C), aiming at a complete treatment of imperative languages. We will show that the logic rules of Section 3 are satisfied by the model, and thus the logic is *sound* ('not self-contradictory').

The view of a program in the model is as the set of its possible traces. Only the terminating traces are of interest, and in consequence the semantics marks out as equals a program that can do nothing, and a program that can do nothing but execute an unending internal loop. So what the logic says about a program is true only supposing control ever reaches that point. NRBG(E) can be viewed as based on a modified CSP trace model [Hoa78] and the standard CSP trace model is recovered when terminating traces all terminate 'normally' in the nomenclature developed here.

Definitions

AD3 A trace is a sequence of 'coloured' atomic transitions between states s from a fixed set S. A transition is written $s \overset{\iota}{\mapsto} s'$ where the states $s, s' \in S$ and the colour $\iota \in \{\mathbf{N}, \mathbf{R}, \mathbf{B}, \mathbf{G}_l, \mathbf{E}_k \mid l \in L, k \in K\}$, where L is a set of labels that may appear in the program and K is a set of exception types.

AD4 A trace set is a subset T of $(S \times S \times \{\mathbf{N}, \mathbf{R}, \mathbf{B}, \mathbf{G}_l, \mathbf{E}_k \mid l \in L, k \in K\})^*$, the set of *finite* sequences of transitions, satisfying:

(a) the first non-**N**-coloured transition ends the sequence;

(b) consecutive transitions $s_{n-1} \overset{\iota n}{\mapsto} s_n$ and $s_n \overset{\iota n+1}{\mapsto} s_{n+1}$ in a trace have a common intermediate state s_n.

AD5 The set of trace sets is $\mathcal{T}(S)$, a complete partial order under subset ordering.

Table A1: Grammar of the abstract imperative language C, where integer variables $x \in X$, term expressions $e \in E$, boolean expressions $b \in B$, labels $l \in L$, exceptions $k \in K$, statements $c \in C$, integer constants $\kappa \in \mathcal{Z}$, infix binary relations $r \in R$, subroutine names $h \in H$.

$$
\begin{aligned}
C &::= \mathbf{skip} \mid \mathbf{return} \mid \mathbf{break} \mid \mathbf{goto}\ l \mid c;c \mid x{=}e \mid b{\rightarrow}c \mid c \triangleleft c \mid \mathbf{do}\ c \mid c:l \mid \mathbf{label}\ l.c \mid \mathbf{call}\ h \\
&\quad \mid \mathbf{try}\ c\ \mathbf{catch}(k)\ c \mid \mathbf{throw}\ k \\
E &::= \kappa \mid x \mid \kappa * e \mid e + e \\
B &::= \top \mid \bot \mid e\ r\ e \mid b \vee b \mid b \wedge b \mid \neg b \\
R &::= < \mid > \mid \leq \mid \geq \mid = \mid \neq
\end{aligned}
$$

Examples

AX1 The simplest example of a trace set is that for a **skip** statement, being the set $\{s \overset{\mathbf{N}}{\mapsto} s \mid s \in S\}$. Running **skip** consists of making a single trivial atomic transition, from a state to the same state again, coloured **N**. The statement exits 'normally', having executed a single **N**-coloured transition.

AX2 A **return** statement has the set of traces $\{s \overset{\mathbf{R}}{\mapsto} s \mid s \in S\}$. I.e., it exits at once 'via a return flow' after a single, trivial, **R**-coloured transition.

We will skip ahead a little by setting out how sequences of statements are handled in the model: the set of traces t of a sequence $a;b$ of programs consists of all the traces t of a that end in a non-**N**-coloured transition plus those traces t that are formed as the concatenation of a trace t_1 of a that ends in a **N**-coloured transition with a trace t_2 of b. I.e. the set of traces of $a;b$ is

$$
\begin{aligned}
&\{s_0 \overset{\iota_1}{\mapsto} \ldots \overset{\iota_n}{\mapsto} s_n \in a \mid \iota_n \neq \mathbf{N}\} \\
\cup\ &\{s_0 \overset{\iota_1}{\mapsto} \ldots \overset{\iota_n}{\mapsto} s_n \mid s_0 \overset{\iota_0}{\mapsto} \ldots \overset{\iota_m}{\mapsto} s_m \in a,\ \iota_m = \mathbf{N},\ s_m \overset{\iota_{m+1}}{\mapsto} \ldots \overset{\iota_n}{\mapsto} s_n \in b\}
\end{aligned}
$$

So there are two ways that a trace from $a;b$ can end 'abnormally' (i.e., in a non-**N**-coloured transition): either it is a trace from a that terminates abnormally, or it is a trace from a that ends normally concatenated with a trace from b that ends abnormally.

There is exactly one way that a trace from $a;b$ can end in a normal (i.e., **N**-coloured) transition: it is composed as a trace from a that ends normally concatenated with a trace from b that ends normally.

AX3 The traces of **skip**;**return** are the set $\{s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{R}}{\mapsto} s \mid s \in S\}$ consisting of two trivial transitions, the last of which is marked as **R**-coloured.

AX4 The traces of **return**;**skip** are the set $\{s \overset{\mathbf{R}}{\mapsto} s \mid s \in S\}$ of just the abnormally-ending traces of **return** (which are all of its traces). There is no trace that can be formed as the concatenation of a trace t_1 from **return** with a trace t_2 from **skip**, because none of the former end in a **N**-coloured transition.

A.1. Model

To permit the model for the NRBG(E) logic to be set out formally, the syntax of an abstract programming language C is given in Table A1. The real programming language C is mapped onto it: a conventional **if**(b) c_1 **else** c_2 statement in C is written as the nondeterministic choice between two guarded statements $b \rightarrow c_1 | \neg b \rightarrow c_2$ in the abstract language; the conventional **while**(b) c loop in C is expressed as **do**$\{\neg b \rightarrow$

Table A2: Standard evaluation of integer and boolean terms of C, for variables $x \in X$, integer constants $\kappa \in \mathcal{Z}$, using primitives $\mathrm{ev}(x, s)$ for evaluation of variable x in a state s and $s[x \leftarrow v]$ for assignment of value $v \in \mathcal{Z}$ to variable x in state s.

$$
\begin{array}{rcl}
[\![-]\!]_- & : & E \times S \to \mathcal{Z} \\
[\![x]\!]s & = & \mathrm{ev}(x, s) \\
[\![\kappa]\!]s & = & \kappa \\
[\![\kappa * e]\!]s & = & \kappa * [\![e]\!]s \\
[\![e_1 + e_2]\!]s & = & [\![e_1]\!]s + [\![e_2]\!]s
\end{array}
\qquad
\begin{array}{rcl}
[\![-]\!]_- & : & B \times S \to \mathbf{bool} \\
[\![\top]\!]s & = & \top \\
[\![\bot]\!]s & = & \bot \\
[\![e_1 < e_2]\!]s & = & [\![e_1]\!]s < [\![e_2]\!]s \\
[\![b_1 \vee b_2]\!]s & = & [\![b_1]\!]s \vee [\![b_2]\!]s \\
[\![b_1 \wedge b_2]\!]s & = & [\![b_1]\!]s \wedge [\![b_2]\!]s \\
[\![\neg b]\!]s & = & \neg([\![b]\!]s)
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{ev} & : & X \times S \to \mathcal{Z} \\
-[- \leftarrow -] & : & S \times X \times \mathcal{Z} \to S \\
\mathrm{ev}(y, s[x \leftarrow v]) & = & \text{if } y = x \text{ then } v \text{ else } \mathrm{ev}(y, s)
\end{array}
$$

break; $c\}$, using the forever-loop of C, etc. A sequence $a; l : b$ in C with a label in the middle is properly expressed as $a : l; b$ in C, but we regard $a; l : b$ in C as syntactic sugar for $a : l; b$, so it is still permissible to write it as $a; l : b$ in C. As a very special sweetener, we permit $l : b$ too, even when there is no preceding statement a, regarding it as an abbreviation for $\phi : l; b$ where ϕ is a statement that has empty trace semantics.

The terms of the abstract language are linear integer forms in integer variables, and the boolean expressions are combinatorial forms in comparisons of terms. Curly brackets may be used to group code statements for clarity, and parentheses may be used to group expressions. The variables are globals and are not formally declared.

Example AX5 A valid integer term is '5x + 4y + 3', and a boolean expression is '$5\mathrm{x} + 4\mathrm{y} + 3 < \mathrm{z} - 4 \wedge \mathrm{y} \leq \mathrm{x}$'.

Remark AR1 The limited set of terms in C makes it difficult to map C-like assignments as simple as '$\mathrm{x} = \mathrm{x} * \mathrm{y}$' or '$\mathrm{x} = \mathrm{x} \mid \mathrm{y}$' (the bitwise or), but in those cases the assignment can be mapped to '$\mathrm{x} = \mathrm{z}$' where the variable z is newly introduced (a so-called 'logical' variable, as opposed to a 'program' variable) and guarded by assertions such as '$(\mathrm{x} \geq 0 \wedge \mathrm{y} \geq 0 \vee \mathrm{x} \leq 0 \wedge \mathrm{y} \leq 0) \to \mathrm{z} \geq 0$'.

In any case, it is rare to map C local variables literally. Usually one maps an abstraction – how many times the variable has been read since last written, for example, which maps $\mathrm{x} = \mathrm{x} * \mathrm{y}$ to $\mathrm{x} = \mathrm{x} + 1; \mathrm{y} = \mathrm{y} + 1; \mathrm{x} = 0$.

The terms of C have the standard evaluation on states as integers and booleans, as shown in Table A2. The evaluation depends on a primitive lookup operation $\mathrm{ev}(x, s)$ for variables $x \in X$, states $s \in S$. We also suppose that there is a primitive assignment operation $s[x \leftarrow v]$ that writes the value $v \in \mathcal{Z}$ to variable x in the state s. If the states are modelled as partial functions $X \nrightarrow \mathcal{Z}$, then $\mathrm{ev}(x, s) = s(x)$ and $s[x \leftarrow v] = s'$ where $s'(y) = \text{if } x = y \text{ then } v \text{ else } s(y)$.

The **label** construct of the abstract language declares $l \in L$ that may be used in **goto**s. A label that has not been declared may not be mentioned in the code. Nor may the same label be declared again in the context of the first.

Given sets $g_l \in \mathcal{T}(S)$ comprising those traces which reach to a **goto** l statement somewhere in the whole program, we interpret a code fragment a as a set of traces $[\![a]\!]_g$

Table A3: Interpretation of programs of language C as trace sets, given as hypothesis the sets of traces $g_l \in \mathcal{T}(S)$ for $l \in L$ as the sequences of transitions that lead to **goto** l statements. A recursive reference means 'the least trace set satisfying the condition'. For $h \in H$, the subroutine named h has code $[h]$.

$$
\begin{aligned}
[\![-]\!]_g &: C \to \mathcal{T}(S) \\
[\![\mathbf{skip}]\!]_g &= \{s \overset{\mathbf{N}}{\mapsto} s \mid s \in S\} \\
[\![\mathbf{return}]\!]_g &= \{s \overset{\mathbf{R}}{\mapsto} s \mid s \in S\} \\
[\![\mathbf{break}]\!]_g &= \{s \overset{\mathbf{B}}{\mapsto} s \mid s \in S\} \\
[\![\mathbf{goto}\ l]\!]_g &= \{s \overset{\mathbf{G}_l}{\mapsto} s \mid s \in S\} \\
[\![\mathbf{throw}\ k]\!]_g &= \{s \overset{\mathbf{E}_k}{\mapsto} s \mid s \in S\} \\
[\![a; b]\!]_g &= \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![a]\!]_g \mid \iota \neq \mathbf{N}\} \\
&\quad \cup \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \mid s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{N}}{\mapsto} s_m \in [\![a]\!]_g,\ s_m \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![b]\!]_g\} \\
[\![x = e]\!]_g &= \{s \overset{\mathbf{N}}{\mapsto} s' \mid s, s' \in S,\ s' = s[x \leftarrow [\![e]\!]s]\} \\
[\![p \to a]\!]_g &= \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![a]\!]_g \mid [\![p]\!]s_0\} \\
[\![a \diamond b]\!]_g &= [\![a]\!]_g \cup [\![b]\!]_g \\
[\![\mathbf{do}\ a]\!]_g &= \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{N}}{\mapsto} s_n \mid s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{B}}{\mapsto} s_n \in [\![a]\!]_g\} \\
&\quad \cup \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \mid s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![a]\!]_g,\ \iota \neq \mathbf{N}, \mathbf{B}\} \\
&\quad \cup \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \mid s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{N}}{\mapsto} s_m \in [\![a]\!]_g,\ s_m \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![\mathbf{do}\ a]\!]_g\} \\
[\![a : l]\!]_g &= [\![a]\!]_g \\
&\quad \cup \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{N}}{\mapsto} s_n \mid s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{G}_l}{\mapsto} s_n \in g_l\} \\
[\![\mathbf{label}\ l.a]\!]_g &= [\![a]\!]_{g \cup \{l \mapsto g_l^*\}} \backslash \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{G}_l}{\mapsto} s_n \mid s_0, \ldots, s_n \in S\} \\
&\quad \text{where } g_l^* = \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{G}_l}{\mapsto} s_n \in [\![a]\!]_{g \cup \{l \mapsto g_l^*\}}\} \\
[\![\mathbf{call}\ h]\!]_g &= \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{N}}{\mapsto} s_n \mid s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![[h]\!]_{\{\,\}},\ \iota \in \{\mathbf{R}, \mathbf{E}_k \mid k \in K\}\} \\
[\![\mathbf{try}\ a\ \mathbf{catch}(k)\ b]\!]_g &= \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \mid s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![a]\!]_g,\ \iota \neq \mathbf{E}_k\} \\
&\quad \cup \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \mid s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{E}_k}{\mapsto} s_m \in [\![a]\!]_g,\ s_m \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![b]\!]_g\}
\end{aligned}
$$

as shown in Table A3. These are traces starting at the beginning of a. They either terminate normally at the end of a, or at some return, break, return, goto or exception in a. The g_l supply an 'initial guess' at the traces to labels l of a coming in via **goto** l statements from elsewhere. The fragment a cannot yield up this information on its own – some of the contributions will come from **goto** l statements not contained in it. The 'initial guesses' will be improved via a fixpoint iteration (the '**label**' equation of Table A3) until they are just right.

Example AX6 Consider the code

$$\mathbf{label}\ A, B.\ \overbrace{\underbrace{\mathbf{skip};\ \mathbf{goto}\ A;\ B:\ \mathbf{return};\ A}_{y}:\ \mathbf{goto}\ B}^{x}$$

with labels A, B, body x, and marked fragment y. The traces of the body x and corresponding code sequences are:

$$
\begin{aligned}
&s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_A}{\mapsto} s && \#\ \mathbf{skip};\ \mathbf{goto}\ A; \\
&s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_B}{\mapsto} s && \#\ \mathbf{skip};\ \mathbf{goto}\ A; A:\ \mathbf{goto}\ B
\end{aligned}
$$

$$s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{R}}{\mapsto} s \qquad \# \textbf{ skip; goto } A; A: \textbf{ goto } B; B: \textbf{ return}$$

with observed trace sets $g_A = \{s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_A}{\mapsto} s\}$, $g_B = \{s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_B}{\mapsto} s\}$ at the labels.

The **goto** B statement is not in the fragment y so there is no way of knowing about the traces g_B while examining y. The indigenous traces of y are only

$$s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_A}{\mapsto} s \qquad \# \textbf{ skip; goto } A$$
$$s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \qquad \# \textbf{ skip; goto } A; A:$$

with no possible entries at label B from within y. That is, the interpretation of y assuming no entries from outside y is $[\![y]\!]_{\{\}} = \{s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_A}{\mapsto} s,\ s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s\}$.

When we hypothesise $g_B = \{s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_B}{\mapsto} s\}$ for y, then y has more traces:

$$s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{R}}{\mapsto} s \qquad \# \textbf{ skip; goto } A; A: \textbf{ goto } B; B: \textbf{ return}$$

corresponding to these entries at B from the rest of the code proceeding to the **return** in y, and $[\![y]\!]_g = \{s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_A}{\mapsto} s,\ s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s,\ s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{R}}{\mapsto} s\}$. In the context of this code, that is the full answer for y. There are no more traces leaving from y.

Example AX7 Consider the code of Example AX6 again. The set $\{s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_A}{\mapsto} s,\ s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_B}{\mapsto} s,$ $s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{R}}{\mapsto} s\}$ is the interpretation of x with assumptions g_A, g_B of Example AX6, and the same g_A, g_B are observed at the labels under these assumptions. This g is the fixpoint g^* of the **label** rule in Table A3.

That rule says to remove hanging traces ending at **goto** As and Bs because they can go nowhere else, leaving only traces $\{s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{R}}{\mapsto} s\}$ exiting from the full code, corresponding to **skip**; **goto** $A; A:$ **goto** $B; B:$ **return**.

Lemma AL1 The constructions shown in Table A3 are trace sets in $\mathcal{T}(S)$ that are monotonic increasing with respect to the g_l, and preserve increasing limits.

Proof. By structural induction on the code a of $[\![a]\!]_g$ in Table A3, and in the case of **label** also by induction on the size of the domain of g. To get through both the **label** and **do** cases, we depend on the fact that the least fixed point in one argument of a monotonic increasing continuous function of two arguments is itself a monotonic increasing continuous function in the remaining argument. □

The simple case is always when the sets $g_l = \{\ \}$ are empty. In that case $[\![a]\!]_g$ comprises just those traces that end at returns, breaks, exceptions and gotos, plus those that reach the end of a normally. In particular, $[\![a]\!]_{\{\}}$ computes traces right up to the point of transition through a **goto**, and no further.

The semantics of the forever **do** loop in Table A3 merits comment. The only way of exiting the loop normally is by exiting the body of the loop via **break**. However, an abnormal exit from the body other than a break, such as a return, does exit the whole loop. Since the fixpoint computation takes place in $\mathcal{T}(S)$, the result is always a set of finite sequences. If the loop never exits, then the fixpoint works out as the empty set of traces, since there are no infinite traces in the sets of the domain and an infinite trace is the only possible trace.

Definition AD6 When all of the labels in the code a are declared in a, we take as canonical interpretation of the code the set of traces $[\![a]\!] = [\![a]\!]_{\{\ \}}$. That is $[\![a]\!]_g$ where g is the partial function with empty domain.

It is often the case that all labels are declared 'up front' in an abstract program. The semantic computation in this case can be organised advantageously:

Proposition AP2 Let $g_\perp = g \cup \{l \mapsto \{\ \}\}$ be the initially empty set assignments for the labels l that appear undeclared in a. The fixpoint semantics for $[\![\textbf{label } l.a]\!]_g$ is, apart from 'extra' traces that all end in a $\mathbf{G}_l$-coloured transition:

$$[\![a]\!]_{g^*_\perp} = [\![a]\!]_{g_\perp} \cup [\![a]\!]_{g'_\perp} \cup [\![a]\!]_{g''_\perp} \cup \ldots$$

where

$$g'_l = \{s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\mathbf{G}_l}{\mapsto} s_n \in [\![a]\!]_g\} \tag{A1}$$

picks out the traces ending in a $\mathbf{G}_l$-coloured transition from $[\![a]\!]_g$, and $g^*_\perp$ is the least fixpoint of $g \mapsto g'$.

Proof. The semantics of **label** given in Table A3 sets $[\![\textbf{label } l.a]\!]_g$ as $[\![a]\!]_{g^*_\perp}$ minus the traces that end in a $\mathbf{G}_l$-coloured transition. Lemma AL1 established that $[\![a]\!]_g$ is monotonic increasing and continuous with respect to the g_l. By Kleene's fixpoint theorem on the complete partial order $\mathcal{T}(S)$,

$$\begin{aligned}[\![a]\!]_{g^*_\perp} &= [\![a]\!]_{g_\perp \cup g'_\perp \cup g''_\perp \cup \ldots} \\ &= [\![a]\!]_{g_\perp} \cup [\![a]\!]_{g'_\perp} \cup [\![a]\!]_{g''_\perp} \cup \ldots\end{aligned}$$

since $g^*_\perp = g_\perp \cup g'_\perp \cup g''_\perp \cup \ldots$ and the sequence $[\![a]\!]_{g_\perp}$, $[\![a]\!]_{g'_\perp}$, $[\![a]\!]_{g''_\perp}$, ... is increasing and bounded above by $[\![a]\!]_{g^*_\perp}$ hence must be $[\![a]\!]_{g^*_\perp}$, since $[\![a]\!]_g$ is monotonic increasing in g and preserves limits. □

The interpretation $[\![a]\!]_{g^*_\perp}$ includes all traces that go through gotos and that is the only difference with respect to $[\![\textbf{label } l.a]\!]$.

Example AX8 Proposition AP2 shows that a goto-self loop has the same semantics as a trivial forever loop:

$$[\![\textbf{label } l.\ l : \textbf{goto } l]\!] = [\![\textbf{do skip}]\!]$$

The computations leading to $[\![l : \textbf{goto } l]\!]_{g^*_\perp}$ are:

$$\begin{aligned}[\![l : \textbf{goto } l]\!]_{g_\perp} &= \{s \overset{\mathbf{G}_l}{\mapsto} s \mid s \in S\} \\ [\![l : \textbf{goto } l]\!]_{g'_\perp} &= \{s \overset{\mathbf{G}_l}{\mapsto} s, s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_l}{\mapsto} s \mid s \in S\} \\ [\![l : \textbf{goto } l]\!]_{g''_\perp} &= \{s \overset{\mathbf{G}_l}{\mapsto} s, s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_l}{\mapsto} s, s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_l}{\mapsto} s \mid s \in S\} \\ &\ldots \\ [\![l : \textbf{goto } l]\!]_{g^*_\perp} &= \{s \overset{\mathbf{G}_l}{\mapsto} s, s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_l}{\mapsto} s, s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{N}}{\mapsto} s \overset{\mathbf{G}_l}{\mapsto} s, \cdots \mid s \in S\}\end{aligned}$$

Throwing away the traces which end in a $\mathbf{G}_l$-coloured transaction gives:

$$[\![\textbf{label } l.\ l : \textbf{goto } l]\!] = \{\ \}$$

The calculation for a busy do-loop yields $[\![\textbf{do skip}]\!] = \{\ \}$ directly. The only conceivable traces are infinite and so do not appear.

Table A4: Extending the language B of propositions to modal operators $\mathbf{N}$, $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_l$, $\mathbf{E}_k$ for $l \in L$, $k \in K$. The interpretation on traces is given for $b \in B$, $b^* \in B^*$.

$$B^* ::= b \mid \mathbf{N}\, b^* \mid \mathbf{R}\, b^* \mid \mathbf{B}\, b^* \mid \mathbf{G}_l\, b^* \mid \mathbf{E}_k\, b^* \mid b^* \vee b^* \mid b^* \wedge b^* \mid \neg b^*$$

$$\begin{aligned}
[\![b]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) &= [\![b]\!]s_n \\
[\![\mathbf{N}\, b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) &= \iota = \mathbf{N} \wedge [\![b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) \\
[\![\mathbf{R}\, b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) &= \iota = \mathbf{R} \wedge [\![b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) \\
[\![\mathbf{B}\, b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) &= \iota = \mathbf{B} \wedge [\![b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) \\
[\![\mathbf{G}_l\, b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) &= \iota = \mathbf{G}_l \wedge [\![b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) \\
[\![\mathbf{E}_k\, b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n) &= \iota = \mathbf{E}_k \wedge [\![b^*]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n)
\end{aligned}$$

Remark AR2 What is the difference between $g^*_\bot$ and $g^*_\top$, where $g_\top = \{l \mapsto [\![\mathbf{G}_l \top]\!]\}$ is the greatest possible starting point for the fixpoint calculation? It is that $g_\top$ injects all possible initial traces at label l, whereas $g_\bot$ injects no traces beyond those arising naturally through the program.

We extend the propositional language B to B^* which includes the modal operators $\mathbf{N}$, $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_l$, $\mathbf{E}_k$ for $l \in L$, $k \in K$, as shown in Table A4. Table A4 also defines an interpretation of B^* on traces. The modal operators of B^* satisfy the algebraic laws given in Table A5. Additionally, however, for $p \in B$,

$$p \;\Leftrightarrow\; \mathbf{N}p \vee \mathbf{R}p \vee \mathbf{B}p \vee \mathbb{W}\, \mathbf{G}_l p \,\mathbb{W} \mathbf{E}_k p \tag{A2}$$

because each transition must be some colour, and those are all the colours.

Proposition AP3 Every $p \in B^*$ can be (uniquely) expressed as

$$p \;\Leftrightarrow\; \mathbf{N}p_{\mathbf{N}} \vee \mathbf{R}p_{\mathbf{R}} \vee \mathbf{B}p_{\mathbf{B}} \vee \mathbb{W}\, \mathbf{G}_l p_{\mathbf{G}_l} \,\mathbb{W} \mathbf{E}_k p_{\mathbf{E}_k}$$

for some $p_{\mathbf{N}}$, $p_{\mathbf{R}}$, etc that are free of modal operators.

Proof. Equation (A2) gives the result for $p \in B$. The rest is by structural induction on p, using the laws of Table A5 and boolean algebra. Uniqueness follows because $\mathbf{N}p_{\mathbf{N}} \;\Leftrightarrow\; \mathbf{N}p'_{\mathbf{N}}$, for example, applying $\mathbf{N}$ to both possible decompositions, and applying the orthogonality and idempotence laws. Apply the definition of $\mathbf{N}$ in the trace model of Table A4 to deduce $p_{\mathbf{N}} \;\Leftrightarrow\; p'_{\mathbf{N}}$ for non-modal predicates $p_{\mathbf{N}}$, $p'_{\mathbf{N}}$. Similarly for $\mathbf{B}$, $\mathbf{R}$, $\mathbf{G}_l$, $\mathbf{E}_k$. □

Remark AR3 Thus modal formulae p may be viewed as tuples of non-modal formulae $(p_{\mathbf{N}}, p_{\mathbf{R}}, p_{\mathbf{B}}, p_{\mathbf{G}_l}, p_{\mathbf{E}_k})$. That means that $\mathbf{N}p \vee \mathbf{R}q$, for example, is simply a convenient notation for writing down two assertions at once: one that asserts p of the final states of the traces that end in a normal transition, and one that asserts q on the final states of the traces that end in a return transition.

Definition AD7 Let $g_l = [\![\mathbf{G}_l\, p_l]\!]$ via the interpretation on traces. Then the interpretation of $\mathbf{G}_l\, p_l \triangleright \{p\}\ a\ \{q\}$, for $p \in B$, $a \in C$ and $p_l, q \in B^*$, is:

$$[\![\mathbf{G}_l\, p_l \triangleright \{p\}\ a\ \{q\}]\!] \;\Leftrightarrow\; [\![\{p\}\ a\ \{q\}]\!]_g$$

Table A5: Laws of the modal operators $\mathbf{N}$, $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_l$, $\mathbf{E}_k$ with $M, M_1, M_2 \in \{\mathbf{N}, \mathbf{R}, \mathbf{B}, \mathbf{G}_l, \mathbf{E}_k \mid l \in L, k \in K\}$ and $M_1 \neq M_2$.

$$
\begin{aligned}
M(\bot) &= \bot && \text{(flatness)}\\
M(b_1 \vee b_2) &= M(b_1) \vee M(b_2) && \text{(disjunctivity)}\\
M(b_1 \wedge b_2) &= M(b_1) \wedge M(b_2) && \text{(conjunctivity)}\\
M(Mb) &= Mb && \text{(idempotence)}\\
M_2(M_1 b) = M_1(b) \wedge M_2(b) &= \bot && \text{(orthogonality)}
\end{aligned}
$$

$$
\Leftrightarrow \quad \forall s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n \in [\![a]\!]_g \mid [\![p]\!] s_0.\ [\![q]\!](s_0 \overset{\mathbf{N}}{\mapsto} \ldots \overset{\iota}{\mapsto} s_n)
$$

That is read as 'the triple $\{p\}\ a\ \{q\}$ holds under assumptions $\mathbf{G}_l\, p_l$ at **goto** l when every trace of a that starts at a state satisfying p terminates in a transition satisfying q'. The explicit Gentzen-style assumptions p_l are free of modal operators. Or they may as well be – see Remark AR3, as applied to $\mathbf{G}_l\, p_l$. What is meant by the notation is that those states that may be attainable as the program traces pass through **goto** statements are assumed to be restricted to those that satisfy p_l.

The $\mathbf{G}_l\, p_l$ among the assumptions are read independently for different l. They may be separated by commas, as $\mathbf{G}_{l_1}\, p_{l_1}, \mathbf{G}_{l_2}\, p_{l_2}, \ldots$, with $l_1 \neq l_2$, etc. Or they may be written as a disjunction $\mathbf{G}_{l_1}\, p_{l_1} \vee \mathbf{G}_{l_2}\, p_{l_2} \vee \ldots$. By Remark AR3, the information in this modal formula is only the mapping $l_1 \mapsto p_{l_1}$, $l_2 \mapsto p_{l_2}$, etc. The number of the disjuncts is governed by the set of undeclared labels l in the code a. If the same l appears twice among the disjuncts $\mathbf{G}_l\, p_l$, then we understand that the union of the two p_l is intended.

Proposition AP4 The following algebraic relations hold:

$$
\begin{aligned}
[\![\{\bot\}\ a\ \{q\}]\!]_g &\Leftrightarrow \top && \text{(A3)}\\
[\![\{p\}\ a\ \{\top\}]\!]_g &\Leftrightarrow \top && \text{(A4)}\\
[\![\{p_1 \vee p_2\}\ a\ \{q\}]\!]_g &\Leftrightarrow [\![\{p_1\}\ a\ \{q\}]\!]_g \wedge [\![\{p_2\}\ a\ \{q\}]\!]_g && \text{(A5)}\\
[\![\{p\}\ a\ \{q_1 \wedge q_2\}]\!]_g &\Leftrightarrow [\![\{p\}\ a\ \{q_1\}]\!]_g \wedge [\![\{p\}\ a\ \{q_2\}]\!]_g && \text{(A6)}
\end{aligned}
$$

for $p, p_1, p_2 \in B$, $q, q_1, q_2 \in B^*$, $a \in C$, and $g_l \subseteq g'_l$.

Proof. By applying Definition AD7. □

It follows, on considering the cases $p_1 \vee p_2 = p_2$ and $q_1 \wedge q_2 = q_1$, that the well-known laws of strengthening and weakening hold:

$$
\begin{aligned}
(p_1 \to p_2) \wedge [\![\{p_2\}\ a\ \{q\}]\!]_g &\Rightarrow [\![\{p_1\}\ a\ \{q\}]\!]_g && \text{(A7)}\\
(q_1 \to q_2) \wedge [\![\{p\}\ a\ \{q_1\}]\!]_g &\Rightarrow [\![\{p\}\ a\ \{q_2\}]\!]_g && \text{(A8)}
\end{aligned}
$$

Moreover, if we consider a different set of hypotheses g' with $g_l \subseteq g'_l$, then

$$
[\![\{p\}\ a\ \{q\}]\!]_{g'} \Rightarrow [\![\{p\}\ a\ \{q\}]\!]_g \qquad \text{(A9)}
$$

The reason is that g'_l is a bigger set than g_l, so $[\![a]\!]_{g'}$ is a bigger set of traces than $[\![a]\!]_g$ by Lemma AL1, and thus the universal quantifier in Definition AD7 produces a smaller (less true) truth value.

Theorem AT1 (*Soundness*) The following algebraic inequalities hold, for $\mathcal{E}_1$ any of $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_l$, $\mathbf{E}_k$; $\mathcal{E}_2$ any of $\mathbf{R}$, $\mathbf{G}_l$, $\mathbf{E}_k$; $\mathcal{E}_3$ any of $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_{l'}$ for $l' \neq l$, $\mathbf{E}_k$; $\mathcal{E}_4$ any of $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_l$, $\mathbf{E}_{k'}$ for $k' \neq k$; $[h]$ the code of the subroutine called h:

$$\left.\begin{array}{r}[\![\{p\}\, a\, \{\mathbf{N}q \vee \mathcal{E}_1 x\}]\!]_g \\ \wedge\, [\![\{q\}\, b\, \{\mathbf{N}r \vee \mathcal{E}_1 x\}]\!]_g\end{array}\right\} \;\Rightarrow\; [\![\{p\}\, a\,;b\, \{\mathbf{N}r \vee \mathcal{E}_1 x\}]\!]_g \tag{A10}$$

$$[\![\{p\}\, a\, \{\mathbf{B}q \vee \mathbf{N}p \vee \mathcal{E}_2 x\}]\!]_g \;\Rightarrow\; [\![\{p\}\, \mathbf{do}\ a\, \{\mathbf{N}q \vee \mathcal{E}_2 x\}]\!]_g \tag{A11}$$

$$\top \;\Rightarrow\; [\![\{p\}\, \mathbf{skip}\, \{\mathbf{N}\, p\}]\!]_g \tag{A12}$$

$$\top \;\Rightarrow\; [\![\{p\}\, \mathbf{return}\, \{\mathbf{R}\, p\}]\!]_g \tag{A13}$$

$$\top \;\Rightarrow\; [\![\{p\}\, \mathbf{break}\ \{\mathbf{B}\, p\}]\!]_g \tag{A14}$$

$$\top \;\Rightarrow\; [\![\{p\}\, \mathbf{goto}\ l\, \{\mathbf{G}_l\, p\}]\!]_g \tag{A15}$$

$$\top \;\Rightarrow\; [\![\{p\}\, \mathbf{throw}\ k\, \{\mathbf{E}_k\, p\}]\!]_g \tag{A16}$$

$$[\![\{b \wedge p\}\, a\, \{q\}]\!]_g \;\Rightarrow\; [\![\{p\}\, b \rightarrow a\ \{q\}]\!]_g \tag{A17}$$

$$[\![\{p\}\, a\ \{q\}]\!]_g \wedge [\![\{p\}\, b\ \{q\}]\!]_g \;\Rightarrow\; [\![\{p\}\, a \,|\, b\ \{q\}]\!]_g \tag{A18}$$

$$\top \;\Rightarrow\; [\![\{q[e/x]\}\ x{=}e\ \{\mathbf{N}q\}]\!]_g \tag{A19}$$

$$[\![\{p\}\ a\ \{q\}]\!]_g \wedge g_l \subseteq [\![\mathbf{G}_l\, q]\!] \;\Rightarrow\; [\![\{p\}\ a : l\ \{q\}]\!]_g \tag{A20}$$

$$[\![\{p\}\ a\ \{\mathbf{G}_l p_l \vee \mathbf{N}q \vee \mathcal{E}_3 x\}]\!]_{g \cup \{l \mapsto \mathbf{G}_l p_l\}} \;\Rightarrow\; [\![\{p\}\ \mathbf{label}\ l.a\ \{\mathbf{N}q \vee \mathcal{E}_3 x\}]\!]_g \tag{A21}$$

$$[\![\{p\}\ [h]\ \{\mathbf{R}r \vee \mathbf{E}_k x_k\}]\!]_{\{\ \}} \;\Rightarrow\; [\![\{p\}\ \mathbf{call}\ h\ \{\mathbf{N}r \vee \mathbf{E}_k x_k\}]\!]_g \tag{A22}$$

$$\left.\begin{array}{r}[\![\{p\}\ a\ \{\mathbf{N}r \vee \mathbf{E}_k q \vee \mathcal{E}_4 x\}]\!]_g \\ \wedge\, [\![\{q\}\ b\ \{\mathbf{N}r \vee \mathbf{E}_k x_k \vee \mathcal{E}_4 x\}]\!]_g\end{array}\right\} \;\Rightarrow\; [\![\{p\}\ \mathbf{try}\ a\ \mathbf{catch}(k)\ b\ \{\mathbf{N}r \vee \mathbf{E}_k x_k \vee \mathcal{E}_4 x\}]\!]_g \tag{A23}$$

Proof. By evaluation, given Definition AD7 and the semantics from Table A3. □

A.2. NRBG(E) logic

On considering a fixpoint g of the map $g \mapsto g'$ described in (A1) of Proposition AP2, the equations (A3-A9) and (A10-A23) of Theorem AT1 give the logical rules of Table A6, via the translation of the semantics of assertions given in Definition AD7. They are sound by construction. What is notable about the fixpoint is that the hypotheses g_l on the left of the '$\triangleright$' cover all the $\mathbf{G}_l g_l$ reached as conclusions on the right. So conclusions cannot be weakened arbitrarily beyond a certain point without weakening hypotheses to match, leading to the restriction in the rule of weakening listed last in Table A6, and in the **go** rule.

A proof must start with a generous guess g_l as to the states that may arise at **goto** l statements, otherwise the **go** rule will not apply. But too large a guess will make the **frm** rule impossible to apply. So the guess has to be 'just right'.

Example AX9 Recall that $l\!:\!a$ is syntactic sugar for $\{\}\!:\!l\,;\, a$ where $\{\}$ is an empty traceset. We derive the rule

$$\frac{\mathbf{G}_l p_l \triangleright \{p \vee p_l\}\ a\ \{q\}}{\mathbf{G}_l p_l \triangleright \{p\}\ l : a\ \{q\}}[\mathrm{frm}_0]$$

as follows:

$$\dfrac{\dfrac{}{\mathbf{G}_l p_l \rhd \{p\}\ \ : l\ \{p \vee p_l\}}\,[\text{frm}] \qquad \mathbf{G}_l p_l \rhd \{p \vee p_l\}\ a\ \{q\}}{\mathbf{G}_l p_l \rhd \{p\}\ \ : l; a\ \{q\}}\,[\text{seq}]$$

Example AX10 The derivation for $\{\top\}$ **label** l. l : **goto** l $\{\bot\}$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{G}_l\top \rhd\ \{\top\}\ \mathbf{goto}\ l\ \{\mathbf{G}_l\top\}}\,[\text{go}]}{\mathbf{G}_l\top \rhd\ \{\top\}\ \mathbf{goto}\ l\ \{\mathbf{N}\bot \vee \mathbf{G}_l\top\}}}{\mathbf{G}_l\top \rhd\ \{\top\}\ l : \mathbf{goto}\ l\ \{\mathbf{N}\bot\}}\,[\text{frm}_0]}}{\mathbf{G}_l\top \rhd\ \{\top\}\ l : \mathbf{goto}\ l\ \{\bot\}}}{\rhd\ \{\top\}\ \mathbf{label}\ l.\ l : \mathbf{goto}\ l\ \{\bot\}}\,[\text{lbl}]$$

The two unlabelled steps are by weakening of the conclusion and/or through the equivalence $\mathbf{N}\bot \leftrightarrow \bot$. If one had tried $\mathbf{G}_l\, p$ for some smaller p, then the first rule could not be applied.

Remark AR4 If we guess $\mathbf{G}_l\top$ and manage to prove a result, then the proof is valid, never mind if the guess is 'just right' or not. In principle, we can go through the proof making the guess smaller (while still above the fixpoint $\mathbf{G}_l\, p_l$, whatever it is) and the steps remain valid. So we do not have to know what the 'just right' guess is if we succeed in proving something from $\mathbf{G}_l\top$.

The model-theoretic justification is that the logic of proof using $\mathbf{G}_l\top$ corresponds to the greatest fixpoint semantics $[\![a]\!]_{g^*_\top}$, not to the least fixpoint semantics $[\![a]\!]_{g^*_\bot}$ (see Remark AR2). But since the greatest fixpoint set of traces includes the least fixpoint set of traces, so what is true of all the greatest fixpoint traces is also true of all the least fixpoint traces.

The logic displayed in Table A6 has to be tailored from C to the real language C. In particular, C has **if** statements instead of guarded statements and non-deterministic choice. Combining the rules **grd** and **dsj** gives the rule for C conditionals, at least when the test expression has no side effects:

$$\dfrac{\rhd\ \{p \wedge c\}\ a\ \{q\} \qquad \rhd\ \{p \wedge \neg c\}\ b\ \{q\}}{\rhd\ \{p\}\ \mathbf{if}(c)\ a\ \mathbf{else}\ b\ \{q\}}\,[\text{if}]$$

When the test has a side-effect, we break the conditional up into an assignment or assignments followed by a conditional with a non-side-effecting test.

C also has **setjmp** and **longjmp** instead of **try**/**catch** and **throw**. The C '**if**(!**setjmp**(k)) a **else** b' corresponds to the C '**try** a **catch**(k) b' construction, and the C '**longjmp**(k,1)' corresponds to the C '**throw** k' construction.

The **call** logic derives from inlining the subroutine body, turning **return**s into '**goto** end' statements, and renaming labels to avoid collisions. But, in practice, calls are all treated as opaque satisfying the property being studied – such as 'balances takes and releases of locks' – while the property is tested for possible failure in each and every subroutine in turn.

Remark AR5 Here is a procedure that, if it terminates, terminates with the code decorated with preconditions and postconditions complying to Table A6.

Start with hypotheses $g_l = \bot$ on the left hand side, and construct preconditions and postconditions throughout the code satisfying the rules of Table A6 with the exception that the restriction on the left of the **go** rule and that in the conditions on

Table A6: Gentzen-style deduction rules for triples of assertions and programs. Unless explicitly noted, assumptions $\mathbf{G}_l p_l$ at left are passed down unaltered from top to bottom of each rule. We let $\mathcal{E}_1$ stand for any of $\mathbf{R}$, $\mathbf{B}$, $\mathbf{G}_l$, $\mathbf{E}_k$; $\mathcal{E}_2$ any of $\mathbf{R}$, $\mathbf{G}_l$, $\mathbf{E}_k$; $\mathcal{E}_3$ any of $\mathbf{R}$. $\mathbf{G}_{l'}$ for $l' \neq l$, $\mathbf{E}_k$; $\mathcal{E}_4$ any of $\mathbf{R}$. $\mathbf{G}_l$, $\mathbf{E}_{k'}$ for $k' \neq k$; $[h]$ the body of the subroutine named h.

$$\frac{\triangleright\ \{p\}\ a\ \{\mathbf{N}q \vee \mathcal{E}_1 x\} \quad \triangleright\ \{q\}\ b\ \{\mathbf{N}r \vee \mathcal{E}_1 x\}}{\triangleright\ \{p\}\ a\,;b\ \{\mathbf{N}r \vee \mathcal{E}_1 x\}}[\text{seq}] \qquad \frac{\triangleright\ \{p\}\ a\ \{\mathbf{B}q \vee \mathbf{N}p \vee \mathcal{E}_2 x\}}{\triangleright\ \{p\}\ \mathbf{do}\ a\ \{\mathbf{N}q \vee \mathcal{E}_2 x\}}[\text{do}]$$

$$\frac{}{\triangleright\ \{p\}\ \mathbf{skip}\ \{\mathbf{N}\,p\}}[\text{skp}] \qquad \frac{}{\triangleright\ \{p\}\ \mathbf{return}\ \{\mathbf{R}\,p\}}[\text{ret}]$$

$$\frac{}{\triangleright\ \{p\}\ \mathbf{break}\ \ \{\mathbf{B}\,p\}}[\text{brk}] \quad [p \to p_l]\frac{}{\mathbf{G}_l\,p_l \triangleright\ \{p\}\ \mathbf{goto}\ l\ \{\mathbf{G}_l\,p\}}[\text{go}]$$

$$\frac{}{\triangleright\ \{p\}\ \mathbf{throw}\ k\ \{\mathbf{E}_k\,p\}}[\text{brw}] \qquad \frac{}{\triangleright\ \{q[e/x]\}\ \ x{=}e\ \ \{\mathbf{N}q\}}[\text{let}]$$

$$\frac{\triangleright\ \{q \wedge p\}\ a\ \{r\}}{\triangleright\ \{p\}\ q \to a\ \{r\}}[\text{grd}] \qquad \frac{\triangleright\ \{p\}\ a\ \{q\} \quad \triangleright\ \{p\}\ b\ \{q\}}{\triangleright\ \{p\}\ a \,;\, b\ \{q\}}[\text{dsj}]$$

$$[p_l \to q]\frac{\mathbf{G}_l\,p_l \triangleright\ \{p\}\ \ a\ \ \{q\}}{\mathbf{G}_l\,p_l \triangleright\ \{p\}\ \ a{:}l\ \ \{q\}}[\text{frm}] \qquad \frac{\mathbf{G}_l\,p_l \triangleright\ \{p\}\ \ a\ \ \{\mathbf{G}_l p_l \vee \mathbf{N}q \vee \mathcal{E}_3 x\}}{\triangleright\ \{p\}\ \ \mathbf{label}\ \ l.a\ \ \{\mathbf{N}q \vee \mathcal{E}_3 x\}}[\text{lbl}]$$

$$\frac{\triangleright\ \{p\}\ \ [h]\ \ \{\mathbf{R}r \vee \mathbf{E}_k x_k\}}{\mathbf{G}_l p_l \triangleright\ \{p\}\ \ \mathbf{call}\ \ h\ \ \{\mathbf{N}r \vee \mathbf{E}_k x_k\}}[\text{sub}] \qquad \frac{\triangleright\ \{p\}\ \ a\ \ \{\mathbf{N}r \vee \mathbf{E}_k q \vee \mathcal{E}_4 x\} \quad \triangleright\ \{q\}\ \ b\ \ \{\mathbf{N}r \vee \mathbf{E}_k x_k \vee \mathcal{E}_4 x\}}{\triangleright\ \{p\}\ \ \mathbf{try}\ \ a\ \ \mathbf{catch}(k)\ \ b\ \ \{\mathbf{N}r \vee \mathbf{E}_k x_k \vee \mathcal{E}_4 x\}}[\text{try}]$$

$$\frac{\triangleright\ \{p_i\}\ \ a\ \ \{q\}}{\triangleright\ \{\curlyvee p_i\}\ \ a\ \ \{q\}} \qquad \frac{\triangleright\ \{p\}\ \ a\ \ \{q_i\}}{\triangleright\ \{p\}\ \ a\ \ \{\curlywedge q_i\}} \qquad \frac{\mathbf{G}_l\,p_{li} \triangleright\ \{p\}\ \ a\ \ \{q\}}{\curlyvee \mathbf{G}_l\,p_{li} \triangleright\ \{p\}\ \ a\ \ \{q\}}$$

$$[p' \to p, q \to q', p'_l \to p_l | \mathbf{G}_l q' \to \mathbf{G}_l p'_l]\frac{\mathbf{G}_l\,p_l \triangleright\ \{p\}\ \ a\ \ \{q\}}{\mathbf{G}_l\,p'_l \triangleright\ \{p'\}\ \ a\ \ \{q'\}}$$

the left of the rule of weakening are ignored. Those relate to final fixpoint conditions g_l^*, not the intermediate conditions g_l of this construction.

Next construct the g'_l that are at least a union of the preconditions found at all **goto** l statements, intersected with the precondition at the statement labelled l. So $g_l \Rightarrow g'_l$ by Lemma AL1. Repeat the construction of preconditions and postconditions satisfying the rules of Table A6 throughout the code, this time with hypotheses g'_l on the left hand side everywhere. Continue until, or if, a fixpoint $g_l = g'_l = g_l^*$ is reached. The fixpoint g_l^* covers all the preconditions of **goto** l statements and fits inside the precondition of the statement labelled l. So the rules of Table A6 are satisfied.

B. A logic specification language

The logical rules are specified individually per analysis. The specification is distributed globally and compiled locally into an analysis tool. Usually only rules for assignments and function calls vary significantly from the Table 1 standard, while the rules for compound statements follow Table 1 exactly. For example, the rule for the 'while(true)' logic in Table 1 is:

$$\frac{\{p\}\ a\ \{\mathbf{B}q \vee \mathbf{N}p \vee \mathcal{E}x\}}{\{p\}\ \mathbf{while(true)}\ a\ \{\mathbf{N}q \vee \mathcal{E}x\}}[\text{whl}]$$

where $\mathcal{E}$ may be any of $\mathbf{R}$, $\mathbf{G}_l$, and it is specified as follows:

```
propn p :: whl(sem a) { propn n,x,q }
```

```
  = (q,x,F)
    where (n,x,q)  =  p :: fix(a) ;
```

The first line (up to the '::') declares the type signature of the rule implementation. It is a transformer of predicates p, by default with result a postcondition (the result type is 'post', if stated explicitly). The type of p is declared as 'propn', signifying a non-modal boolean formula.

The first line also names the transformation 'whl' and declares an argument parameter a of semantic ('sem') type, representing the body of the while loop. It is another transformer of predicates, constructed and provided at runtime. The same line also declares local variables for propositions n, x, q, used internally.

The second line gives the postcondition generated for the while statement, expressed as a $(\mathbf{N}, \mathbf{R}, \mathbf{B})$ triple. It is $\mathbf{N}q \vee \mathbf{R}x \vee \mathbf{B}\bot$, i.e., $\mathbf{N}q \vee \mathbf{R}x$.

The third line gives details of the propositions n, x, q. They satisfy

$$(p \to n), \quad (\{n\}\ a\ \{\mathbf{N}n \vee \mathbf{R}x \vee \mathbf{B}q\})$$

The '$\mathbf{fix}(a)$' transformation generates suitable n, x and q from the input p. There always is a solution (n, x, q) since ($\top$, $\top$, $\top$) will do. The **fix** operation attempts to find a tighter solution than that, however. It does so by relaxing the constraints p to obtain n, examining the structure of p in order to do so. Thus the rule implemented is exactly:

$$[p \to n]\frac{\{n\}\ a\ \{\mathbf{B}q \vee \mathbf{N}n \vee \mathbf{R}x\}}{\{p\}\ \mathbf{while}(\mathbf{true})\ a\ \{\mathbf{N}q \vee \mathbf{R}x\}}[\text{whl}]$$

The rule is extended to cover the case where $\mathbf{G}_l g_l$ appears instead of or in addition to $\mathbf{R}x$ using the '**with**' term and pattern constructor:

```
propn p :: whl(sem a) { propn n,x,q; ctx g }
  = (q,x,F) with g
    where (n,x,q) with g  =  p :: fix(a) ;
```

The indexed set of conditions g_l defining the possible states at the **goto** l exits in the code is declared as type 'ctx' (context). The implemented rule is now

$$[p \to n]\frac{\{n\}\ a\ \{\mathbf{B}q \vee \mathbf{N}n \vee \mathbf{R}x \vee \mathbf{G}_l g_l\}}{\{p\}\ \mathbf{while}(\mathbf{true})\ a\ \{\mathbf{N}q \vee \mathbf{R}x \vee \mathbf{G}_l g_l\}}[\text{whl}]$$

However, the full rule should reference a set of assumptions $\mathbf{G}_l\, h_l$ defining the states attainable at **goto** l statements. It ought to be:

$$[p \to n]\frac{\mathbf{G}_l\, h_l \rhd \{n\}\ a\ \{\mathbf{B}q \vee \mathbf{N}n \vee \mathbf{R}x \vee \mathbf{G}_l g_l\}}{\mathbf{G}_l\, h_l \rhd \{p\}\ \mathbf{while}(\mathbf{true})\ a\ \{\mathbf{N}q \vee \mathbf{R}x \vee \mathbf{G}_l g_l\}}[\text{whl}]$$

That is specified by a reference h of type 'ctx' that is passed down to the computation for the body a on the left, and returned, incremented, as g at right:

```
ctx h, propn p :: whl(sem a) { propn n,x,q; ctx g }
  = (q,x,F) with g
    where (n,x,q) with g  =  h,p :: fix(a) ;
```

Some h_l may be broadened to cover a $\mathbf{G}_l g_l$ that would not otherwise fit within it, as part of the fixpoint calculation procedure outlined in Remark AR5

Defect specifications are composed in three parts. The first part is an initial condition that is imposed at entry to the program. For example:

$$x \leq 0$$

where x is a logical variable (i.e., declared in the specification file, not a program variable). This is written:

```
::propn initial() = (x<=0);
```

in the configuration file. The second part of the specification is the declaration of an objective function that will be computed at syntax tree nodes:

```
propn p ::term objective() = upper[x: p];
```

That means objective$(p) = \sup\{x \mid p\}$, the upper bound of the x such that p.

Certain of the logic rules have 'rule attachments' in C code that check the value of the objective function when the rule is active, setting an entry in a database if the value exceeds a certain level. The attachments are placed within curly braces at the end of the rule. A rule with the attachment

```
  if ([[p :: objective()]] > 0)  setflags(DEFECT, CHAR, LINE);
```

conditionally adds a line and character count to the DEFECT database table. The double square brackets indicate to parse the quoted text as part of the configuration language, rather than C. The quote has 'term' type in the configuration language, which returns an integer to C.

Via re-configuration, the analysis software can run different analyses. Table B7 shows the logic of C from Table 1 rendered into the specification language, and Table B8 shows the extra configuration required to detect 'calls to a function that may sleep under spinlock'.

Table B7: Rules of Table 1, the programming logic for C, expressed in the logic specification language. The formula $\mathbf{N}n \vee \mathbf{R}r \vee \mathbf{B}b = \mathrm{fix}(s, p)$ holds iff $\{n\}s\{\mathbf{N}n \vee \mathbf{R}r \vee \mathbf{B}b\}$ and $p \to n$, i.e., n is a fixpoint of statement s above p. The formula $q = \mathrm{im}(e, x, p)$ holds iff $\forall x.p[x] \to q[e/x]$, i.e., q is the image of p under $x \mapsto e$. The $\vee$ operator denotes logical disjunction of propositions and also pointwise union of indexed vectors of sets. The $h\backslash\backslash l$ operation removes the lth component $h.l$ from the indexed vector h. For brevity, declarations of the local variables defined in `where` clauses have been omitted.

```
ctx h, propn p::skp()                    = (p, F, F) with ctx h;
ctx h, propn p::seq(sem s1, s2)          = (n2, r1∨r2, b1∨b2) with ctx h2;
                                  where (n2,r2,b2) with ctx h2 = h1, n1::s2
                                  and   (n1,r1,b1) with ctx h1 = h,  p ::s1;
ctx h, propn p::if(sem s1, s2)           = (n1∨n2, r1∨r2, b1∨b2) with ctx h1∨h2
                                  where (n1,r1,b1) with ctx h1 = h, p::s1
                                  and   (n2,r2,b2) with ctx h2 = h, p::s2;
ctx h, propn p::whl(sem s)               = (b, r, F) with ctx h1;
                                  where (n,r,b) with ctx h1 = h, p::fix(s);
ctx h, propn p::let(var x; term e)       = (q, F, F) with ctx h
                                  where q = x, p::im(e);
ctx h, propn p::go(label l)              = (F, F, F) with ctx h∨{l::p};
ctx h, propn p::brk()                    = (F, F, p) with ctx h;
ctx h, propn p::ret()                    = (F, p, F) with ctx h;
ctx h, propn p::frm(label l; sem s1, s2) = (n2, r1∨r2, b1∨b2) with ctx h2
                                  where (n1,r1,b1) with ctx h1 = h, p::s1
                                  and   (n2,r2,b2) with ctx h2 = h1, n1∨h.l::s2;
ctx h, propn p::lbl(label l; sem s)      = (n, r, b) with ctx h\\l
                                  where (n,r,b) with ctx h1 = h, p::s;
```

Table B8: Configuration of the logic specification to search for the defect where a function that may sleep is called under lock.

```
              ::propn initial()    = (x≤0);
       propn p::term  objective()  = upper[x:p];
ctx h, propn p::post  unlk()       = (p[x+1/x], F, F) with ctx h;
ctx h, propn p::post  lock()       = (p[x-1/x], F, F) with ctx h;
ctx h, propn p::post  call(name l) = (p, F, F) with ctx h {
 if (sleepy(l) && [[p::objective()]]>0) setflags(DEFECT,CHAR,LINE);
};
```